machine code applications
for the ZX spectrum

expert machine code techniques

david laine

Spectrum Machine Code Applications contains advanced machine code routines to deal with problems such as floating point numbers, output to the screen and animated graphics. All the routines are fully explained and annotated.

Through the application of the host of routines presented the author explains how successful machine code routines are written, tested and used in practical applications.

This is not another introductory book on machine code but an insight into the way a professional machine code programmer looks at the Spectrum.

0946408173

SUNSHINE

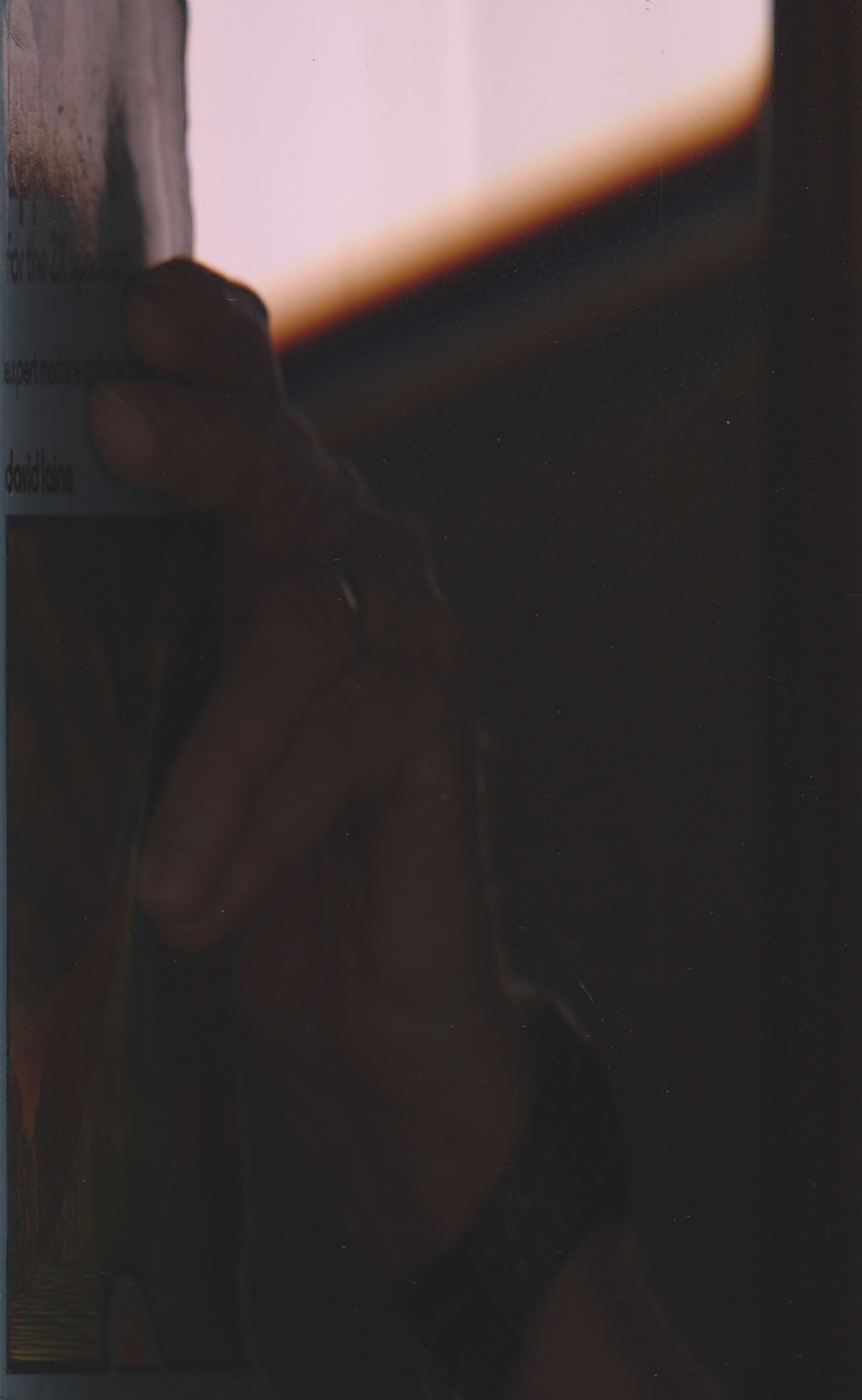
ISBN 0 946408 17 3

£6.95

DAVID LAINE

SPECTRUM MACHINE CODE APPLICATIONS

SUNSHINE



machine code applications for the ZX spectrum

expert machine code techniques

david laine

First published 1983 by:
Sunshine Books
(An imprint of Scot Press Ltd.)
12–13 Little Newport Street,
London WC2R 3LD

ISBN 0 946408 17 3

Cover design by Graphic Design Ltd.
Illustration by Stuart Hughes
Typeset and printed in England by Commercial Colour Press, London E7.

CONTENTS

Contents in detail

To the women in my life, living and dead, without whom I would not have had the strength or encouragement to create the book. Also to my colleagues in London and Malvern who showed me how to go about it.

"Some books are to be tasted, others to be swallowed, and some few to be chewed and digested; that is, some books are to be read only in parts; others to be read but curiously; and some few to be read wholly, and with diligence and attention".

Of Studies
Francis Bacon 1561 – 1626

CHAPTER 1
Introduction

This book is not intended for the absolute beginner, but for someone who has used machine code programs from books or magazines and feels the urge to try his or her hand.

To those of you who are still interested, this book is not a thesis on the instruction code or the internal operations of the Spectrum. If you do not already own one, you will need to obtain a book which explains how the Z80 functions — most things I shall explain, but some things will be omitted through over familiarity or because I did not set out to detail them. I do include a synopsis of the available instructions and their execution times but have not touched on peripheral programming, the interrupt vector register nor the refresh register. My purpose is to present an introduction to machine code programs which can interface with BASIC, which I assume that you already know thoroughly.

Why should you use machine code?

For total freedom from the restraints of BASIC and an increase in speed. I have included an array sort routine (Chapter 10) which is about 125 times faster than its BASIC equivalent (and then show how you can double that). On the other hand the errors make themselves known that much faster.

For the machine code programs I have used a simple assembler by Picturesque.

Always remember that if you can see a logical way of solving a problem then that problem can be solved. The hardest thing for the beginner is sticking it out, the resolution to persevere until the last error is removed and the code runs correctly.

Do not, to begin with, attempt more than one or two hours at a stretch, and do keep notes on your errors. After a few weeks the worst of the nerves will be over and you will have become well acquainted with machine code.

For any problem, write down what you want to do and then draw flow diagrams. If you can't do a bit of the problem put it in a little box and carry on with the main problem; later go back and work on the boxes as if they were full grown problems in their own right.

Finally, never forget: the true programmer exists in one of two states: the depths of despair because the program is not working, or the highest elation because it is known why the program is not working.

CHAPTER 2
About Programming

'An engineer was called from afar; the machine would not work; he pondered the problem; he called for a hammer; he dealt the thing a resounding blow; it worked. Much later the bill arrived:

Transport and travel	£50.00
Hitting the machine	£00.01
Knowing what and where to hit	£500.00
Total	£550.01 (+VAT).'

(Modernised Apocryphal)

Programming is far more an art than a science. Science is involved, for the rules imposed by the machine code instructions and any operating system admit of no flexibility. But the presence of the finest ingredients hardly implies great cuisine if the cook is a gorilla — on the other hand, a great cook can conjure a feast from the most unpromising beginnings.

There is constant interplay amongst eight things:

Reliability
Simplicity
Testability
Speed
Size
Documentation
Program environment
Program specification

Reliability

Djikstra's conjecture:

If a program has N instructions, each having a probability p of doing the right thing, then the probability of the program doing the right thing is of the order of p^N.

If the program is to loop L times, then the probability is of the order of p^{NL}, which means that if p is not equal to 1 then the program is not worth running.

Every fault in a program ought to be investigated, explained and corrected. A faulty program is not worth running, a misplaced comma has cost millions before now.

Simplicity

There is no merit in making programs needlessly convoluted. The whole, no matter how complicated, can always be broken down into a few simple parts and these parts further reduced to simpler parts. I find that a very good way to test a program is to draw lines on a listing from jump instructions to the relevant labels. The results are usually self evident.

Testability

Much has been written about testability; all I shall say here is that simplicity of structure makes testing that much easier. You can have more combinations of bit pattern in a mere 40 bytes than there are atoms in the universe.

Speed

Each instruction takes a finite time to execute and there are always several possible instruction mixes to produce the same result. If you have a piece of program which seems to be slow to produce results, examine it for loops within loops within loops. Improvements in speed may require changes in data structure which may mean that the program becomes bigger.

Size

'Anybody can build a bridge, but only an engineer can, just.'

The size of a program is the sum of its two parts — the instructions and the data area.

Data should never be written into and be part of a program except (perhaps) in test programs. The program should be given a pointer to the location of its data and be allowed to work from there.

The number of instructions can nearly always be reduced. The more straightforward the program construction the easier and more effective the reduction will be.

Documentation

A program or subroutine without proper and adequate documentation might as well not exist. You can retain sufficient memory of a piece of a program for about three months to prompt you, with a listing, as to how and why and what. Beyond those three months the program becomes a liability.

Documentation does not need to become a magnum opus, just:

List of entry conditions a) registers
b) special locations

List of exit conditions a) registers
b) special locations
c) preserved registers
d) flags set

Brief description of the function

These, together with a listing and flow diagram should be kept in a good note book with stiff covers. If you can also keep the original source code on tape so much the better. I use message cassettes myself, they seem to do quite well.

Program environment

A fancy way of saying what extra peripherals you have beyond the TV screen. You must always tailor your output to suit. What looks impressive in flashing, scintillating colour will look very different on a ZX printer.

Program specification

This is left to the end because everything else affects and is affected by it. It may be necessary to go round the whole loop several times to arrive at an acceptable compromise.

There are particular aspects which must be considered if you are producing a program for someone else:

a) Do you understand what he says he wants?
b) Is what he says he wants a true expression of what he needs? Remember that you and he have to have a common appreciation of the problem to be solved.
c) Can you see the problem as one of a more general sort that you have already solved, or, more generally, have you solved something like it already? Is this problem going to be the first of a series? Would it be better to write a more general program for future needs? For example, given the need to integer arithmetic extending over seven bytes, might it not be better to devise general solutions extending over N bytes and then set N to 7 for the specific case?
d) If the problem is a large one, time spent designing the data base can repay vast dividends in time needed to extract data. All the data referring to a major item should be stored together so that it can be got at through a single page register. Different settings of the page register are then used to point to different data items.

e) When you have a solution scheme worked out you will also have some questions to ask, so go back to a) above and start again.

CHAPTER 3
Instructions

The instruction codes and their actions on the flags are given condensed form in **Figure 3.1** and **Figure 3.2** together with their allowed address combinations. These tables are no substitute for the books mentioned in Chapter 1.

Form of **Figure 3.1**

column	description
1	operation mnemonic
2	symbolic operation
3	allowed address combinations (where two addresses are allowed the two groups of possibles are separated by a space).

The numbers under some of the addresses indicate the execution times of the associated operation (in computer clock cycles).

N	indicates that a 1 byte value may be used
NN	indicates that a 2 byte value may be used
(NN)	indicates that the address of a byte is to be used
d	is a 1 byte page offset to be used with a page register
DISP	is the displacement to a nearby instruction

The stack

The stack is a concertina-like list which stores items in a first-in first-out (FIFO) form. It is like a pile of cards — the first one you place on top is the first to be removed, but to confuse matters it is held in memory 'upside- down'. The top of the stack (ie where the last item added is) is at a lower address than the bottom (ie where the very first item lies). The Stack Pointer SP is a 16-bit register which points to the address of the last item on the stack.

Normally the stack is used for storing return addresses from subroutines, in the form of a pair of bytes, and a CALL puts a pair on the stack (and decrements SP by two), and a RET will remove it (and increment SP by two). However, there are two other types of instructions that use the stack — PUSH and POP. When a 16-bit register is PUSHed

Figure 3.1a

<table>
<tr><th>CODE</th><th>OPERATION</th><th>ADDRESSES - CONDITION CODES - REGISTERS AND EXECUTION TIMES (CLOCK CYCLES)</th></tr>
<tr><td>ADC</td><td>A ← A + S + Cy
HL ← HL + S + Cy</td><td>A   A|B|C|D|E|H|L|N|(HL)|(IX + d)|(IY + d)
←— 4 —→←7→←— 19 —→
HL   BC|DE|HL|SP
←— 15 —→</td></tr>
<tr><td>ADD</td><td>A ← A + S
HL ← HL + S
IX ← IX + S
IY ← IY + S</td><td>A   A|B|C|D|E|H|L|N|(HL)
←— 4 —→←7→
HL   BC/DE|HL|SP
←— 11 —→
IX   BC/DE|IX|SP
←— 11 —→
IY   BC|DE|IY|SP
←— 11 —→</td></tr>
<tr><td>AND</td><td>A ← A ∧ S</td><td>A|B|C|D|E|H|L|N|(HL)|(IX+d)|(IY+d)
←— 4 —→←7→←— 19 —→</td></tr>
<tr><td>BIT</td><td>Z_F ← b + S</td><td>0|1|2|3|4|5|6|7   A|B|C|D|E|H|L|(HL)|(IX + d)|(IY + d)
←— 8 —→←12→←— 20 —→</td></tr>
<tr><td>CALL</td><td>STACK PC
PC ← NN</td><td>C|NC|Z|NZ|M|P|PE|PO   NN
←— 17 —→
(10 IF NOT OBEYED)
NN
17</td></tr>
<tr><td>CCF</td><td>Cy ← $\overline{Cy}$</td><td></td></tr>
<tr><td>CP</td><td>FLAGS ← A - S</td><td>A|B|C|D|E|H|L|N|(HL)|(IX+ d)|(IY+d)
←— 4 —→←7→←— 19 —→
SEE ALSO CPD CPDR
F3.1d CPI CPIR</td></tr>
<tr><td>CPL</td><td>A ← $\overline{A}$</td><td>4</td></tr>
<tr><td>DAA</td><td colspan="2">RESULT ADJUST FOR USE WITH BCD ARITHMETIC</td></tr>
<tr><td>DEC</td><td>S ← S - 1</td><td>A|B|C|D|E|H|L|BC|DE|HL|SP|IX|IY|(HL)|(IX+d)|(IY+d)
←— 4 —→←— 6 —→←10→←11→←— 23 —→</td></tr>
<tr><td>DI</td><td colspan="2">DISABLE INTERRUPTS - SPECTRUM USES NON MASKABLE INTERRUPTS</td></tr>
<tr><td>DJNZ</td><td>B ← B - 1
B ≠ 0 JR NN
B = 0 NOP</td><td>DISPLACEMENT
13 OBEYED
8 NOT OBEYED
SEE JR BELOW</td></tr>
<tr><td>EI</td><td colspan="2">ENABLE INTERRUPTS - SPECTRUM USES NON-MASKABLE INTERRUPTS</td></tr>
<tr><td>EX</td><td>(SP) ⇆ S
AF ⇆ AF'
DE ⇆ HL</td><td>(SP)   HL|IX|IY
←19→←23→
AF   AF'
←4→
DE   HL
←4→</td></tr>
<tr><td>EXX</td><td>BC ⇆ BC'
DE ⇆ DE'
HL ⇆ HL'</td><td>4</td></tr>
<tr><td>HALT</td><td></td><td></td></tr>
<tr><td>IM</td><td>SET
INTERRUPT
MODE</td><td>0|1|2
←8→</td></tr>
<tr><td>IN</td><td>S ← INPUT (C)
A ← INPUT (N)</td><td>A|B|C|D|E|H|L   (C)
←— 12 —→
A   (N)
11
SEE ALSO IND INDR
F3.1d INI INIR
READ PERIPHERAL PORT</td></tr>
<tr><td>INC</td><td>S = S + 1</td><td>A|B|C|D|E|H|L|BC|DE|HL|SP|IX|IY|(HL)|(IX + d)|(IY + d)
←— 4 —→←— 6 —→←10→←11→←— 23 —→</td></tr>
</table>

Figure 3.1b

JP	PC ← S	(HL)|(IX)|(IY)|NN ←4→←8→←10→		
	PC ← S IF CC	C|NC|Z|NZ|M|P|PE|PO ←10→	NN	
JR	PC ← PC + DISP	DISPLACEMENT ←12→	ABS (DISP) ≤ 127 PC POINTS TO NEXT OP	
	PC ← PC + DISP IF CC	C|NC|Z|NZ ←12→ OR 7 IF NOT OBEYED		
LD		A	(BC)|(DE)|(HL)|(NN)|(IX+d)|(IY+d) ←7→←13→←19→	
		A|B|C|D|E|H|L	A|B|C|D|E|H|L|N ←4→ 10	
		A|B|C|D|E|H|L	(HL)|(IX+d)|(IY+d) ←7→←19→	
		BC|DE|HL|SP	NN|(NN) 10→ 20→	SEE ALSO F3,1d LDD LDI LDDR LDIR
		IX|IY	NN|(NN) 14→ 20→	
	LOAD 1ST ADDRESS WITH 2ND ADDRESS			
		SP	HL|IX|IY 6→ 10→	
		(HL)	A|B|C|D|E|H|L|N ←7→ 10	
		(BC)|(DE)|(HL)	A 7	
		(IX+d)|(IY+d)	A|B|C|D|E|H|L|N ←19→	
		(NN)	A|BC|DE|HL|IX|IY|SP 13 ←20→	
		I/R	A 9	
NEG	A = −A	[illegible]		
NOP	NOTHING	4		
OR	A = A OR S	A|B|C|D|E|H|L|N|(HL)|(IX+d)|(IY+d) ←4→←7→←19→		
OUT	OUTPUT S TO ADDR. BC	(C)	A|B|C|D|E|H|L ←12→	SEE ALSO F3,1d OUTD OUTI OTDR OTIR
	OUTPUT A TO ADDR. An	(N)	A 11	
POP	READ FROM TOP OF STACK SP = SP + 2	AF|BC|DE|HL|IX|IY ←10→←14→		
PUSH	LOAD ON STACK SP = SP − 2	AF|BC|DE|HL|IX|IY ←10→←14→		
RES	S_b ← 0	0|1|2|3|4|5|6|7	A|B|C|D|E|H|L|(HL)|(IX+d)|(IY+d) ←8→←15→←23→	
RET	POP PC OF STACK	10		
	POP PC IF CC	C|NC|Z|NZ|M|P|PO|PE ←11→ OBEYED, 5 NOT OBEYED		
RET I	RETN FROM INTERRUPT			
RETN	RET I FROM NON MASKABLE INTERRUPT			
RL	ROTATE LEFT	A|B|C|D|E|H|L|(HL)|(IX+d)|(IY+d) ←8→←15→←23→		SEE F3,2 R1
RLA	ROTATE A LEFT	4		SEE F3,2 R1

Figure 3.1c

<table>
<tr><td>RLC</td><td>ROTATE RIGHT</td><td>A|B|C|D|E|H|L|(HL)|(IX+d)|(IY+d)
←— 8 —→←— 15 —→←— 23 —→</td><td>SEE F3,2 R2</td></tr>
<tr><td>RLCA</td><td>ROTATE A RIGHT</td><td>4</td><td>SEE F3,2 R2</td></tr>
<tr><td>RLD</td><td>ROTATE LEFT
A AND (HL)</td><td>18 BCD A AND (HL) NIBBLES</td><td>SEE F3,2 R3</td></tr>
<tr><td>RR</td><td>ROTATE RIGHT</td><td>A|B|C|D|E|H|L|(HL)|(IX+d)|(IY+d)
←— 8 —→←— 15 —→←— 23 —→</td><td>SEE F3,2 R4</td></tr>
<tr><td>RRA</td><td>ROTATE A RIGHT</td><td>4</td><td>SEE F3,2 R4</td></tr>
<tr><td>RRC</td><td>ROTATE RIGHT</td><td>A|B|C|D|E|H|L|(HL)|(IX+d)|(IY+d)
←— 8 —→←— 15 —→←— 23 —→</td><td>SEE F3,2 R5</td></tr>
<tr><td>RRCA</td><td>ROTATE A RIGHT</td><td>4</td><td>SEE F3,2 R5</td></tr>
<tr><td>RRD</td><td>ROTATE RIGHT
A AND (HL)</td><td>18 BCD A AND (HL) NIBBLES</td><td>SEE F3,2 R6</td></tr>
<tr><td>RST</td><td>STACK PC
PC ← S</td><td>0|8|16|24|32|40|48|56
←— 11 —→</td><td>FOR JUMPS TO HEAD OF ROM</td></tr>
<tr><td>SBC</td><td>A ← A−S−Cy

HL ← HL−S−Cy</td><td>A

HL</td><td>A|B|C|D|E|H|L|N|(HL)|(IX+d)|(IY+d)
←— 4 —→←— 7 —→←— 19 —→
BC|DE|HL|SP
←— 15 —→</td></tr>
<tr><td>SCF</td><td>SET Cy</td><td>4</td><td></td></tr>
<tr><td>SET</td><td>S_b ← 1</td><td>0|1|2|3|4|5|6|7</td><td>A|B|C|D|E|H|L|(HL)|(IX+d)|(IY+d)
←— 8 —→←— 15 —→←— 23 —→</td></tr>
<tr><td>SLA</td><td>SHIFT LEFT
ARITHMETIC</td><td>A|B|C|D|E|H|L|(HL)|(IX+d)|(IY+d)
←— 8 —→←— 15 —→←— 23 —→</td><td>SEE F3,2 S1</td></tr>
<tr><td>SRA</td><td>SHIFT RIGHT
ARITHMETIC</td><td>A|B|C|D|E|H|L|(HL)|(IX+d)|(IY+d)
←— 8 —→←— 15 —→←— 23 —→</td><td>SEE F3,2 S2</td></tr>
<tr><td>SRL</td><td>SHIFT RIGHT
LOGICAL</td><td>A|B|C|D|E|H|L|(HL)|(IX+d)|(IY+d)
←— 8 —→←— 15 —→←— 23 —→</td><td>SEE F3,2 S3</td></tr>
<tr><td>SUB</td><td>A ← A−S</td><td>A|B|C|D|E|H|L|N|(HL)|(IX+d)|(IY+d)
←— 4 —→←— 7 —→←— 19 —→</td><td></td></tr>
<tr><td>XOR</td><td>A ← A ⊕ S</td><td>A|B|C|D|E|H|L|N|(HL)|(IX+d)|(IY+d)
←— 4 —→←— 7 —→←— 19 —→</td><td></td></tr>
</table>

Figure 3.1d

BLOCK AND REPEAT LOAD AND COMPARE

CPD	FLAGS ← A−(HL) HL ← HL−1 BC ← BC−1	COMPARISON CP	ALL COMPARISON INSTRUCTIONS LEAVE THE A REGISTER UNALTERED BUT JUST SET THE FLAGS ACCORDING TO THE SUBTRACTION. D OR I REFERS TO DECREMENTING OR INCREMENTING THE CONTENS OF HL. IN ALL CASES THE BC REGISTER PAIR IS DECREMENTED
CPI	HL ← HL+1 BC ← BC−1		
CPIR	HL ← HL−1 BC ← BC−1	REPEAT UNTIL A = (HL) OR BC = Ø	
CPDR	HL ← HL+1 BC ← BC−1	REPEAT UNTIL A = (HL) OR BC = Ø	
LDD	(DE) ← (HL) DE ← DE−1 HL ← HL−1 BC ← BC−1	LOAD	
LDI	DE ← DE+1 HL ← HL+1 BC ← BC−1		
LDDR	DE ← DE−1 HL ← HL−1 BC ← BC−1	REPEAT UNTIL BC = Ø	DECREMENT AFTER EXECUTION
LDIR	DE ← DE+1 HL ← HL+1 BC ← BC−1	REPEAT UNTIL BC = Ø	DECREMENT AFTER EXECUTION
		BLOCK AND REPEAT INPUT/OUTPUT	
IND	(HL) ← (C) B ← B−1 HL ← HL−1	BC CONTAINS (INPUT) PORT ADDRESS,	HL CONTAINS DATA ADDRESS
INI	B ← B−1 HL ← HL+1		
INDR	B ← B−1 HL ← HL−1	REPEAT UNTIL B = Ø	DECREMENT BEFORE EXECUTION
INIR	B ← B−1 HL ← HL+1	REPEAT UNTIL B = Ø	DECREMENT BEFORE EXECUTION
OUTD	(C) ← (HL) B ← B−1 HL ← HL−1	BC CONTAINS (OUTPUT) PORT ADDRESS,	HL CONTAINS DATA ADDRESS
OUTI	B ← B−1 HL ← HL+1		
OTDR	B ← B−1 HL ← HL−1	REPEAT UNTIL B = Ø	DECREMENT BEFORE EXECUTION
OTIR	B ← B−1 HL ← HL+1	REPEAT UNTIL B = Ø	DECREMENT BEFORE EXECUTION

Figure 3.2

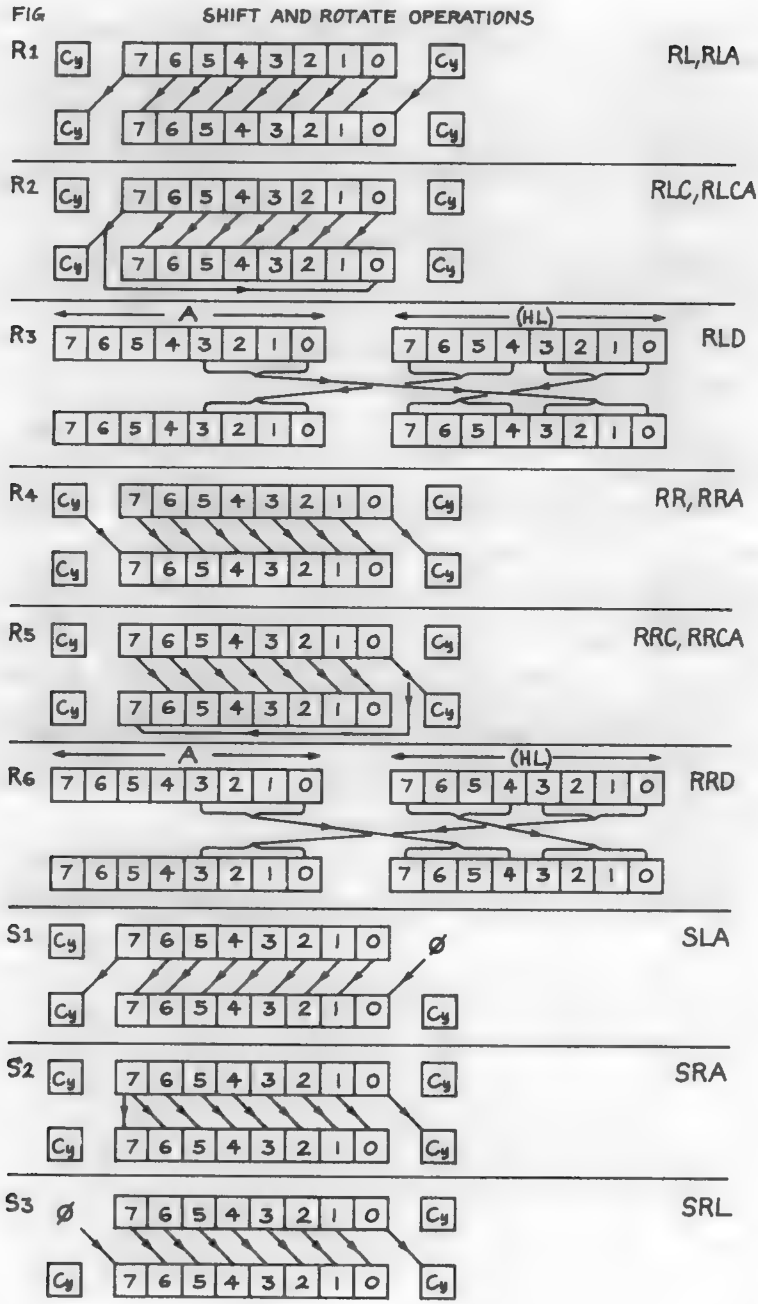


FIG
SHIFT AND ROTATE OPERATIONS
R1
RL,RLA
R2
RLC,RLCA
R3
A
(HL)
RLD
R4
RR,RRA
R5
RRC, RRCA
R6
RRD
S1
SLA
S2
SRA
S3
SRL

Figure 3.3

FLAGS (IN F REGISTER)

FLAG SETTING INSTRUCTIONS		S	Z		H		P/V	N	Cy	COMMENTS
ARITHMETIC										
8 BIT	ADD A, r; ADC A, r	*	*		*		V	Ø	*	
	SUB r ; SBC r	*	*		*		V	1	*	
	CP r	*	*		*		V	1	*	FLAGS SET FOR A − r
	NEG	*	*		*		V	1	*	FLAGS SET FOR A = −A
16 BIT	ADD							Ø	*	
	ADC	*	*				V	Ø	*	
	SBC	*	*				V	1	*	
LOGICAL	AND r	*	*		1		P	Ø	Ø	
	OR r ; XOR r	*	*		Ø		P	Ø	Ø	
	CPL				1			1		A = $\overline{A}$
ROTATE	RLA ; RRA				Ø			Ø	*	ROTATE A
	RLCA ; RRCA				Ø			Ø	*	ROTATE A AND Cy
	RL r ; RR r	*	*		Ø		P	Ø	*	ROTATE R
	RLC r ; RRC r	*	*		Ø		P	Ø	*	ROTATE R AND Cy
SHIFT	SLA r ; SRA r	*	*		Ø		P	Ø	*	
	SRL r	*	*		Ø		P	Ø	*	
BIT TEST	BIT b, r		*		1			Ø		BIT B OF r PLACED IN Z
I/O TRANSFER	IN r, (C)	*	*		Ø		P	Ø		
	INI ; IND	$	*					1		BLOCK I/O
	OUTI ; OUTD	$	*					1		$ = Ø IF B ≠ Ø
	INIR ; INDR	$	1					1		$ = 1 IF B = Ø
	OTIR ; OTDR	$	1					1		
BLOCK										
MOVE	LDI ; LDD				Ø		#	Ø		# SET IF BC = 1
	LDIR ; LDDR				Ø		Ø	Ø		
SEARCH	CPI ; CPD		$		*		#	1		$ SET IF A = (HL)
	CPIR ; CPDR		$		*		#	1		# SET IF BC = 1
OTHERS	CCF							Ø	$	Cy = $\overline{Cy}$
	DAA	*	*		*		P		*	ADJUST RESULT TO CONTINUE BCD ARITHMETIC
	DEC r	*	*		*		V	1		
	INC r	*	*		*		V	Ø		
	LD A, I	*	*		Ø		$	Ø		THE INTERRUPT ENABLE FLIP
	LD A, R	*	*		Ø		$	Ø		FLOP IS MOVED TO P/V
	RLD ; RRD	*	*		Ø		P	Ø		LEFT AND RIGHT BCD ROTATE
	SCF				Ø			Ø	1	

NOTES ON THE TABLE

1. NO SYMBOL – NO ACTION
2. [redacted] UNSET
3. 1 SET
4. P P/V SET ACCORDING TO PARITY OF RESULT
5. V P/V SET AS A RESULT OF OVER- OR UNDER-FLOW
6. [redacted] MAY BE SET OR UNSET
7. b IS A BIT NUMBER Ø (1) 7
8. r A SINGLE REGISTER OR A BYTE VALUE
9. $ } SEE ADJACENT COMMENT
10. # }

its two bytes are put on the top of the stack, and SP decremented by two. The opposite is POP which places the values of the top two bytes on the stack into a 16-bit register. SP is then incremented by two.

Repeated PUSHing will eventually reduce SP until it starts to overwrite your program or data, and unfunny things will start to happen, usually resulting in a system reset.

As long as POPS and PUSHs are kept in step the SP pointer will not 'run away'; usually 200 or so bytes are sufficient but with deeply nested subroutines and more advanced programming than is dealt with in this book you will perhaps need more. Remember that the higher you set the head address of your program the less room there is for the stack.

If POP and PUSH become unbalanced over a subroutine then, in general, the subroutine cannot exit correctly (a very common beginner's problem). However, if on entry to the subroutine, you store the SP value in some address (which the stack is not going to over-write!), you can always exit correctly from the deepest level of nesting, by resetting SP from the stored value and executing a RET instruction, eg

```
GRAFS   LD    (ADDR),SP
        ..
        ..
EXIT    LD    HL,(ADDR)
        LD    SP,HL
        RET
```

Everything that was left on the stack still exists but is just abandoned and will be overwritten by subsequent PUSH operations.

CHAPTER 4
Number Representation

'When *I* use a word,' Humpty Dumpty said, in a rather scornful tone 'it means just what I choose it to mean — neither more nor less.'
'The question is,' said Alice, 'Whether you can make words mean so many different things.'
'The question is,' said Humpty Dumpty, 'which is to be master — that's all.'

(*Alice through the Looking Glass*)

Given the content of any byte very little can be said about it except its value. Its meaning depends on the programmer or program which gave the byte that particular value.

Example 1

If the byte is a copy of the F register (FLAGS) then you must refer to **Figure 3.3** and even then you may need to work back through the program to determine which operation on what data set a particular bit.

Example 2

It may be part of a Spectrum standard floating point number (**see Figure 4.1**). Before you can assign a meaning to the byte you must determine which of the five possible bytes it is.

Example 3

It may be one byte of a 16 bit integer — again which byte?

Example 4

It may be a genuine byte value such as an ASCII character code or a Spectrum token, in which case the meaning can be determined by inspection of **Figure 4.2**. Note that when and if you use an RS232/V24 type of interface you will almost certainly need to insert transmission control codes and may also be required to set or unset the MS bit of each ASCII character according to the parity required by the peripheral.

Figure 4.1

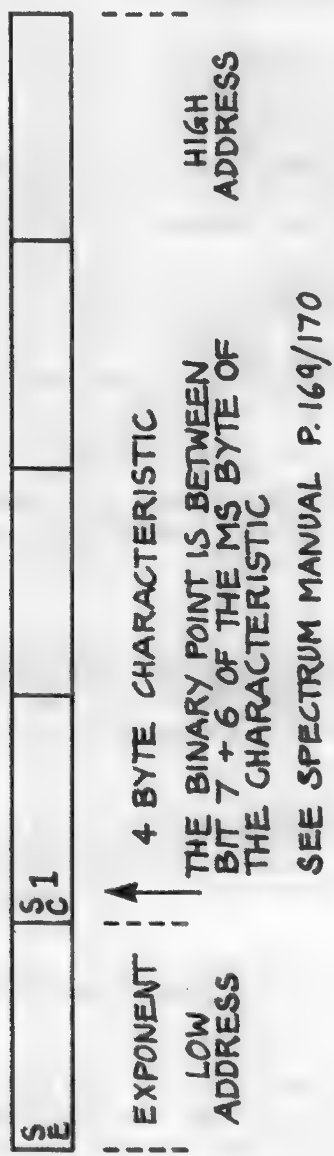

S E
S C 1
EXPONENT
LOW ADDRESS
4 BYTE CHARACTERISTIC
THE BINARY POINT IS BETWEEN BIT 7 + 6 OF THE MS BYTE OF THE CHARACTERISTIC
SEE SPECTRUM MANUAL P. 169/170
HIGH ADDRESS

Figure 4.2

BYTE VALUE INTERPRETATION

BYTE							
7	6	5	4	3	2	1	0
TABLE		ROW			COLUMN		

ASCII CODES

	0	1	2	3	4	5	6	7
0	NUL	SOH	STX	ETX	EDT	EN	ACK	BEL
1	BS	HT	LF	VT	FF	CR	SO	S1
2	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
3	CAN	EM	SUB	ESC	FS	GS	RS	ITB
4	SPACE	!	* "	* #	$	%	&	'
5	(	)	*	+	,	–	•	/
6	Ø	1	2	3	4	5	6	7
7	8	9	:	;	<	=	>	?

TABLE Ø

	0	1	2	3	4	5	6	7
0	@	A	B	C	D	E	F	G
1	H	I	J	K	L	M	N	O
2	P	Q	R	S	T	U	V	W
3	X	Y	Z	* [	* \	*]	* _	* ↑
4	£	a	b	c	d	e	f	g
5	h	i	j	k	l	m	n	o
6	p	q	r	s	t	u	v	w
7	x	y	z	* {	* |	* }	* ~	* ©

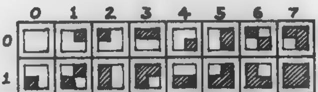
TABLE 1

* THESE CHARACTERS ARE NATIONAL VARIABLES

SPECTRUM TOKENS

	0	1	2	3	4	5	6	7
0							PRINT	EDIT
1	⇦	⇨	⇩	⇧	DELETE	ENTER	NUMBER	
2	INK	PAPER	FLASH	BRIGHT	INVERSE	OVER	AT	TAB
3								

TABLE Ø



	0	1	2	3	4	5	6	7
2	a	b	c	d	e	f	g	h
3	i	j	k	l	m	n	o	p
4	q	r	s	t	u	RND	INKEY$	PI
5	FN	POINT	SCREEN$	ATTR	AT	TAB	VAL$	CODE
6	VAL	LEN	SIN	COS	TAN	ASN	ACS	ATN
7	LN	EXP	INT	SQR	SGN	ABS	PEEK	IN

USER DEFINED GRAPHICS

TABLE 2

	0	1	2	3	4	5	6	7
0	USR	STR$	CHR$	NOT	BIN	OR	AND	<=
1	>=	< >	LINE	THEN	TO	STEP	DEF FN	CAT
2	FORMAT	MOVE	ERASE	OPEN	CLOSE	MERGE	VERIFY	BEEP
3	CIRCLE	INK	PAPER	FLASH	BRIGHT	INVERSE	OVER	OUT
4	LPRINT	LLIST	STOP	READ	DATA	RESTORE	NEW	BORDER
5	CON-TINUE	DIM	REM	FOR	GOTO	GOSUB	INPUT	LOAD
6	LIST	LET	PAUSE	NEXT	POKE	PRINT	PLOT	RUN
7	SAVE	RANDOM-ISE	IF	CLS	DRAW	CLEAR	RETURN	COPY

TABLE 3

Example 5

It may be (part of) an instruction code. If you start off in the wrong place the result will be gibberish.

There is no way of knowing, from the byte alone, what it is. If, however, the location of the start of the machine code program can be determined the rest of the program follows logically and in running is all sorted out by the hardware.

Floating point numbers (see Figure 4.1)

Read the Spectrum manual pages 169–170. What follows is a note on manipulating fp numbers.

The Sign of the characteristic is in the lowest addressed byte.

When working with fp numbers, always adjust the size of the exponent such that the bit after the characteristic sign bit is the inverse of the sign bit; that is, the characteristic begins either 01 or 10, never 00 or 11.

To add or subtract fp numbers, first adjust the exponents to be the same (shift the characteristic of the lower fp number right as its exponent is increased) then add or subtract the characteristics as required and correct the exponent for over- or under-flow if need be. This shifting to equalise exponents is known as normalisation.

Multiplication and division of fp numbers

1 Don't, unless you have to.
2 If you must
   a) add or subtract exponents
   b) multiply or divide the characteristics
   or c) get the BASIC to do it for you!

Data structures

Data structures can be as simple or complex, long or short, as you wish, can unravel and can find room to handle. Each set of problems has its own solutions.

Suppose that much alpha-numeric data has to be handled, we have A–z, A–Z, 0–9, space and punctuation. If we introduce a shift character to distinguish between upper and lower case and put digits in the opposite case to punctuation, then the whole can be squeezed into 40 separate codes. Now 40 * 40 * 40 = 64000 and 16 bits in two bytes has a maximum value of 65535. For the price of some coding we can get three characters where there were only two before — an increase of 50% in the available storage.

Again there is another scheme: there are 26 * 26 = 676 letter pair combinations, aa, ab, ac, ... zx, zy, zz, by no means all of which exist in English (or any other language for that matter). It may well be that in a particular application, less than 256 such pairs exist; in such a case the input may be coded at two letters per byte with a resulting doubling of the storage capacity.

If we are handling large arrays of numerical data, whose entries are mostly empty (the so-called sparse arrays) we may have to design techniques for handling the data, not as arrays, but in terms of the non null elements and their locations. This will be slow but at least we will be able to handle the problem.

Signed and unsigned arithmetic

Signed arithmetic uses the MS bit of the value to indicate the arithmetic sign of the remaining bits. In unsigned arithmetic you keep track of the signs of the values of the variables. Usually it suffices to ignore the sign bit as is done in addressing (but keep an eye on the carry flag).

CHAPTER 5
Addressing

Addressing refers to the method by which data or constants stored in memory are read into the Z80 registers, and is a very important concept. The Z80 has many modes, some more useful in certain applications than others.

Be very clear in your own mind whether you are using 8 or 16 bit variables. Addresses are always 16 bit values and refer either to a byte or the lower of the two bytes used for a 16 bit value (but remember that in the BASIC program area the Spectrum system has line numbers swapped around).

There are several methods of getting at data: some are outlined below:

Direct

The location is known and has a name or numerical value.

eg	LD HL,(23626)	will put the contents of 23627/8 into the HL register pair.
	LD A,(23627)	will put the contents of byte 23627 into the accumulator or A register.

Direct + fixed offset

At run time this is identical to the direct method.

eg	LD B,(PHRED + 5)	PHRED is a value determined by the assembler at assembly time.

With most assemblers the address can be generated from any mixture of labels and values together with + and – signs. Also a label may be assigned a value rather than having a value determined for it by the assembler.

Indirect

The address of the required data is held in some known location.

eg	LD H1,(PHRED)	HL is loaded with the address.
	LD B,(HL)	B loaded with the byte addressed by the content of HL.

Page addressing

Page addressing, also known as Indexed Addressing uses two 16 bit registers — IX and IY. A page in this context is an area of not more than 256 bytes whose head address is loaded as a 16 bit value in the IX (or IY) register. There are assumed to be several such pages, all laid out in the same order, each containing data for an individual item — see Chapter 10 for an example. Data is then handled by means of fixed offsets relative to the head of each page.

eg LD A,(IX+5)	will load A with the 6th byte of the page pointed to by the address currently held in IX.

The method becomes more transparent if the fixed offset is given a name indicating the contents. Consider processing examination results. Each student is given a page, organised thus:

BYTE NO.	CONTENTS	
0 } 1 }	Student No.	
2	Marks	Mathematics
3	...	English
4	...	Physics
5	...	
6		
7 ... etc.		

We can then code: LD A,(IX+PHYSICS) so long as we have let the assembler know that PHYSICS has the value 4. To load HL with the student's number we have to code:

```
LD  L,(IX+0)        to load the low order byte
LD  H,(IX+1)        to load the high order byte
```

To move on to the next student we need only add a suitable constant to the page register. Page 180 of the Spectrum manual says that the IY register should not be used, but this is not strictly true. Although its value should *never* be altered, it is always set to 23610, and it can be used to access some of the system variables. For example, to set bit 1 of FLAGS the instruction would be

```
SET 1,(IY+)
```

Multiple indirect

If we have access to only the address of the value, we have to repeat the process used to extract an indirect address. There is no theoretical limit to the depths to which one can sink in this process though I should consider it unreasonable to attempt more than three levels of descent.

Chaining (see Figure 5.1)

This is a method of linking (usually blocks of) data together so that a rapid search can be made. Chaining requires that each data item carries with it the address of one or more related data items. These addresses are known also as pointers. Chaining can be forward, backward or both together. The deletion of an item from a chain is accomplished by pointing around it, an item not pointed to does not exist.

It is usually necessary to produce a 'garbage' collection routine to reorder data and physically remove deleted entries from chained data.

Note that several independent chains can link through the same data (so long as pointer space is supplied).

The Spectrum BASIC program is a part forward chain. Each line carries what amounts to a pointer to the head of the next line. Forward searching is easy, backward searches (such as GOTO... a previous line number) are fresh searches from the beginning.

Computation of addresses (and instructions)

When working with a variety of addresses, it is sometimes tempting to construct the address (or instruction), enter it into the code and then obey it.

This technique is not to be recommended, but may be tolerated, especially where speed and size are of importance. I do use it and all I shall say is 'be careful'. Remember also that you cannot use the technique if the program is going to be loaded into a PROM or ROM.

Notes

1. Only use it in subroutines, never the 'main line' of a program.
2. On entry to the subroutine, be sure that you know what the sate of any computed instruction will be. Never compute an instruction for 'next time round'.
3. Be very aware of how the assembler *you* use assembles the instructions that are modified — some instructions can be assembled in different ways:

eg LD HL,(NN) can be coded (hex)
2A–n1–n2
or EX–6B–n1–n2

Figure 5.1a

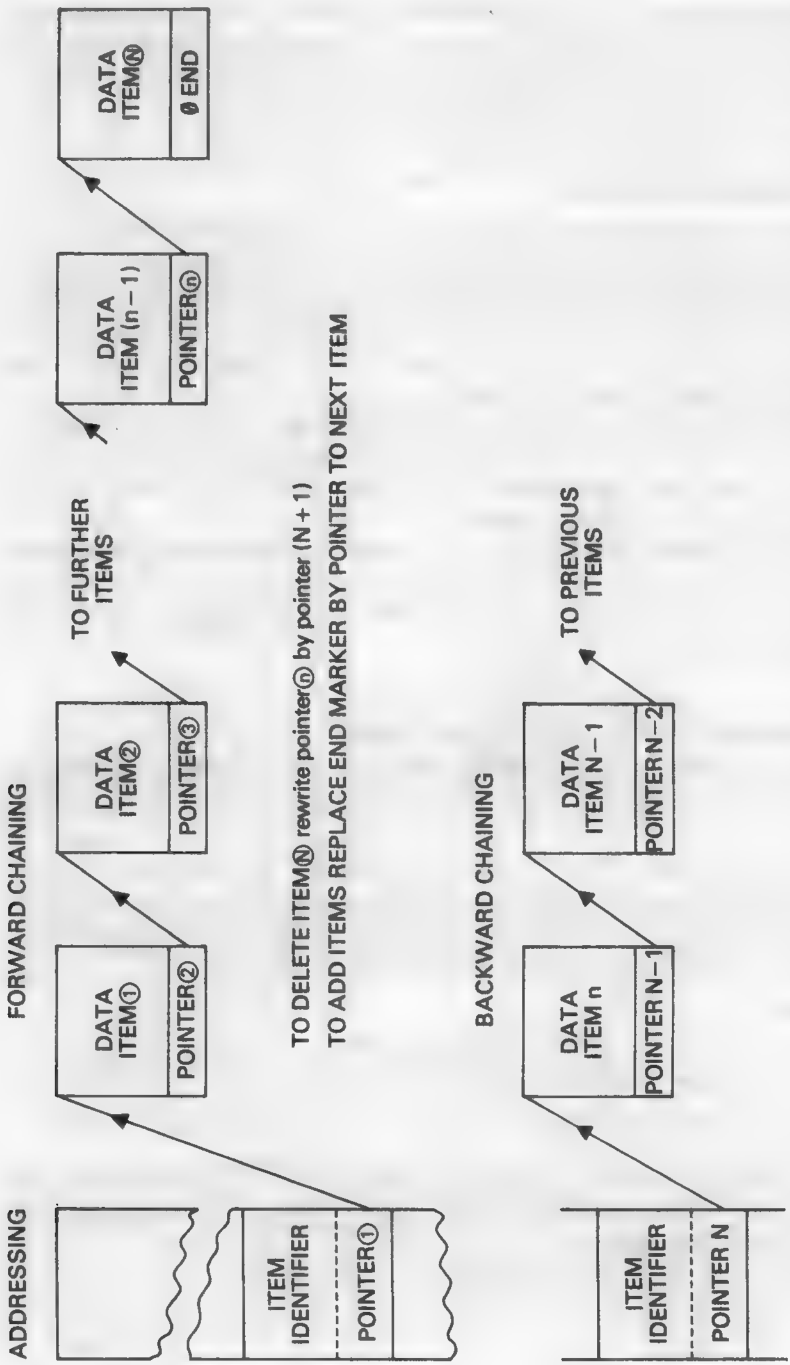
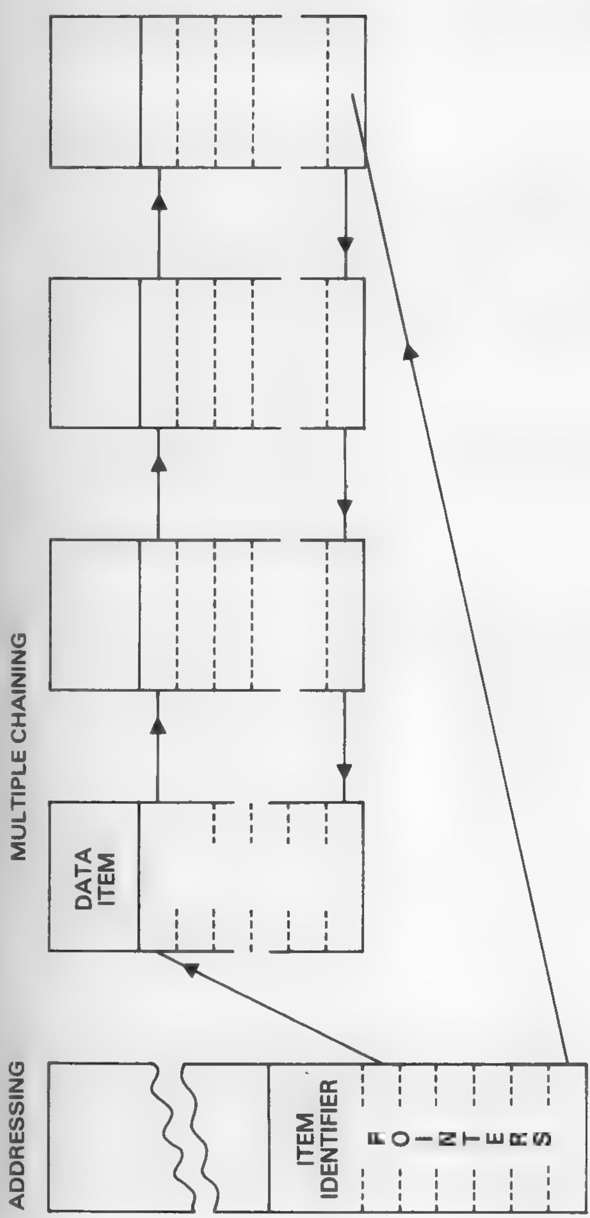

ADDRESSING
ITEM IDENTIFIER
POINTER①
FORWARD CHAINING
DATA ITEM①
POINTER②
DATA ITEM②
POINTER③
TO FURTHER ITEMS
DATA ITEM (n − 1)
POINTER(n)
DATA ITEM(N)
Ø END
TO DELETE ITEM(N) rewrite pointer(n) by pointer (N + 1)
TO ADD ITEMS REPLACE END MARKER BY POINTER TO NEXT ITEM
BACKWARD CHAINING
ITEM IDENTIFIER
POINTER N
DATA ITEM n
POINTER N − 1
DATA ITEM N − 1
POINTER N − 2
TO PREVIOUS ITEMS

Figure 5.1b


ADDRESSING
MULTIPLE CHAINING
DATA ITEM
ITEM IDENTIFIER
POINTERS

(The code examples in this book, to the best of my knowledge, use an assembler which produces the shorter of two equivalent forms.)

4 If you label the instruction, then the label has the address value of the first byte.

5 Remember, when you document or publish the code, to draw particular attention to what you have done. Another person's assembler may use the other assembly option or you may change assembler.

CHAPTER 6
Simple Beginnings

Introduction

This chapter deals by example with two essential aspects of machine code programming; execution times for a piece of program and the passing of information into subroutines. I also attempt to give some insight into the way in which solutions develop. I regret that I know of no way in which years of experience can be grafted into the beginner. As you gain experience look back over your earlier efforts and wince, the more you wince the more you have learned.

Clearing the display buffer; an essay on execution times

An elementary routine to clear the 6144 bytes of the display buffer, starting at 16384.

My first thought was along the lines of:

```
1          LD      BC,6144
2          LD      HL,16384
3  CLRE    LD      A,0
4          LD      (HL),A
5          INC     HL
6          DEC     BC
7          LD      A,B
8          OR      C
9          JR      NZ,CLRE
```

which works, but is most inelegant.

Note, however:

a) Lines 7, 8 and 9 as a means of testing BC = 0 since a double register DEC or INC operation affects no flags.
b) Instructions 3 and 4 can be amalgamated; I forgot that LD (HL),0 is a valid byte instruction.

The loop 3/4 to 9 requires 37 clock cycles (see **Figure 3.1**) and is executed 6144 times to give a requirement of 227300 clock cycles. Can we do it faster?

Version 2

```
        LD      HL,16384
        LD      C,24
LIN3    LD      B,0
LIN4    LD      (HL),0
        INC     HL
        DJNZ    LIN4
        DEC     C
        JR      NZ,LIN3
```

The inner loop, the main time consumer in any such routine, requires 6144*29 = 178000 clock cycles, which is some 78% of the requirement of the first attempt.

There are however problems if and when one wishes to generalise the solution which depends on 6144 being equal to 24*256: and are B and C correctly set up for the DEC and JR operations?

We haven't come to the end of the road yet. What we have tacitly done is to load the same location into successive locations. Suppose we cleared location 0 and then moved location 0 into location 1, and then moved location 1 into location 2 etc. Put another way what happens if we used the LDIR operation:

```
1 LD      HL,16384
2 LD      DE,16385
3 LD      BC,6143
4 LD      (HL),0
5 LDIR
```

and everything is done by LDIR 6143 times at 21 clock cycles a time. The time is thus 129000 cycles, or 57% of the first attempt.

If we attempt to put all the variables into parameters and make a fully fledged subroutine out of this, to be fully general, we will have all the complexities of picking up the parameters. Just now this will be more trouble than it is worth.

With a minor change to line 4 we make a clear display subroutine CLRD (**Listing 6.1**) which we enter with A = 0 and record that all registers are destroyed.

Setting up the attributes area

A straight crib is in order here, just change the values assigned to HL, DE, and BC and give the routine a new name, SETA, which is entered with A = required attributes byte.

Listing 6.1

```
1335 CLRD    PUSH AF
1340         PUSH BC
1345         PUSH HL
1350         LD   HL,16384
1355         LD   DE,16385
1360         LD   BC,6143
1365         LD   (HL),0
1370         LDIR
1375         POP  HL
1380         POP  BC
1385         POP  AF
1390         RET
```

Listing 6.2

```
0745 WAIT$   PUSH BC
0746         PUSH DE
0747         PUSH HL
0748         LD   BC,0
0749         LD   DE,0
0750         LD   HL,0
0751         PUSH AF
0752         LDIR
0753         POP  AF
0754         POP  HL
0755         POP  DE
0756         POP  BC
0757         RET
0760 LWAIT   PUSH BC
0761         LD   B,0
0762 LWAIU   CALL WAIT$
0763         DJNZ LWAIU
0764         POP  BC
0765         RET
```

I make no claim that the routines in this book are anywhere near minimum execution time or minimum length. Two or three people in competition should be able to make significant savings in time and space in most of the subroutines.

Wait and the passing of data into subroutines

When working with a machine code program, it is quite easy to execute output to the display faster than it can be displayed — certainly far faster than it can be comprehended. We need a routine that slows things down.

Going back again to the CLRD routine yet again, the LDIR operation is fairly slow. If it were entered with HL = DE and BC = 0 it would consume 65536*21 clock cycles or about a $\frac{1}{4}$ second at an 8 Mhz clock rate. So we can code the WAIT routine **(Listing 6.2** and **6.3)** taking care to save all the registers and restore them afterwards so that we can insert CALL WAIT at any point we wish without disrupting things.

If we want a still longer wait we can put a call of WAIT inside another loop to get a wait of 60–70 seconds (routine LWAIT).

While we are dealing with the WAIT function, it is often an idea to be able to wait until a key is pressed, and while we are doing this we can set specific key options (for use later on with data entry, cursor movement, games etc.).

The answer to 'how?' is in location 23560 of the Spectrum variables area. This contains the code for the last key pressed. Remember the Spectrum interrupt system is running all the time your routines are working, (you are, in jargon, time sharing with it), so we can just loop, reading 23560 until the code we want appears.

There are two problems to be answered

a) How do we form the list?
b) How do we tell the routine where the list is?

Commentary

The list must contain two things, a character code and an address to be accessed when that character is met. Some assemblers do not allow an address to be put into a list and the address may be so far away that a relative or displacement jump may not be used. An entry in the list must look like:

Character code
JP ADDRESS

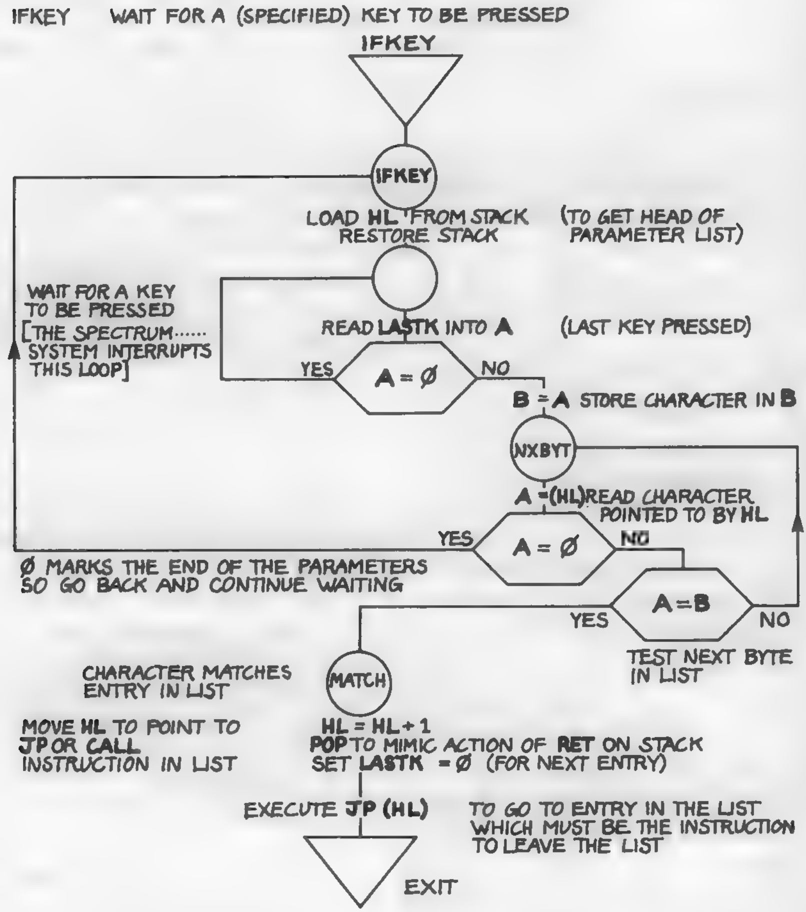
What about the length of the list?

We could work out the length of the list beforehand and pass it into the routine in a register, but if we want to add or delete list entries this must be

Listing 6.3

```
0520 PAUSE   PUSH AF
0525         PUSH BC
0530         PUSH DE
0535         PUSH HL
0540 PAUS1   LD   A,(LASTK)
0545         CP   0
0550         JR   Z,PAUS1
0555         LD   (CHAR$),A
0560         LD   A,0
0565         LD   (LASTK),A
0570         POP  HL
0575         POP  BC
0580         POP  AF
0585         RET
0590 CHAR$   DEFB 0
0595 LASTK   EQU  23560
0600 IFKEY   POP  HL
0605         PUSH HL
0610 IF1     LD   A,(LASTK)
0615         CP   0
0620         JR   Z,IF1
0625         LD   B,A
0630 NXBYT   LD   A,(HL)
0635         CP   0
0640         JR   Z,IFKEY
0645         CP   B
0650         JR   Z,MATCH
0655         LD   DE,4
0660         ADD  HL,DE
0665         JR   NXBYT
0670 MATCH   INC  HL
0675         POP  BC
0680         LD   B,A
0685         LD   A,0
0690         LD   (LASTK),A
0695         JP   (HL)
```

Flowchart 6.1



changed as well. A better way is to sacrifice a character code and mark the end with that. I use 0 as it is unused anyway and is easily tested for.

The list now looks like:

```
Code 1
JP ADDR1
Code 2
JP DDR2
nop
```

A form of the list has been settled, how do we tell the routine where it is?

There are two schools of thought here. One says that lists and suchlike constants should be kept neatly segregated in a section. The other says that, as far as possible, all the constants should be found reasonably close to the routines which require them.

I tend toward the second school in this instance, as any routine becomes more rather than less self documenting. So, if we call the routine IFKEY (which tends to be self explanatory) its use could look like this:

```
CALL  IFKEY
DEFB  "A"
JP    AREAD
DEFB  "+"
JP    INCR
NOP
```

(DEFB puts a character code in the code). The call of IFKEY hangs the program until either the A key is pressed (in capital shift) or the + key is pressed (in symbol shift).

How do we get the list into the routine?

The top of the stack contains the return address of the sub-routine, so it points to the code for 'A' in the above example. To read it we simply pop it off the stack into a suitable register. IFKEY, you will have realised, is called as a subroutine but does *not* return to the calling program through a RET instruction, which requires an extra pop action to match the push action of the CALL operation.

Synopsis

CLRD	clears the display.
IFKEY	waits until one of a preset list of keys is operated.
WAIT	causes a (roughly) ¼ second pause.
LWAIT	causes about a one minute pause.

CHAPTER 7
Display Output

The only real way for the Spectrum to communicate with its user is via the TV display, so it is very important to be able to do this. I will firstly present the necessary calculating routine, followed by a full character output program.

To output to the TV we must first be able to locate a pixel in the display buffer. From this routine we go on to write ASCII characters, display text strings, display octal numbers and report the contents of the registers. Along the way there is an introduction to the idea of 'global variables'.

PLOT: locating a pixel in the display buffer

'The display file stores the television picture. It is rather curiously laid out...' Spectrum manual Chapter 24 p 164.

In all these routines the origin of the display is *the top lefthand corner* of the display area.

The display is divided into three sections, each of eight text lines (64 lines of pixels). There are 256 pixels per row — 32 bytes hold the data for 1 row, a bit set is an ink dot. The next 32 bytes after those for row 0 hold the data for row 8, and the next 32 byte block holds the data for row 16 and so on for the first third of the screen (see **Figure 7.1a**).

From this we can deduce that a horizontal position (or x coordinate) specifies a single bit in one of the 256 bits of a 32 byte block.

Stage 1 is then to take the x value in one byte and then use the three least significant bits to point to a bit in some byte. The remaining five most significant bits specify which byte in the 32 byte block is involved.

Stage 2 is to determine which of the 192 blocks of 32 bytes is involved. This must be deducted from the vertical position (or y coordinate). From **Figure 7.1a** we see that:

Row 0 uses block 0
Row 1 uses block 8
Row 2 uses block 16 etc.

This may not give much inspiration, written like this, but remember that we are dealing with a computer and if we think in binary or octal we may be better off.

Figure 7.1a

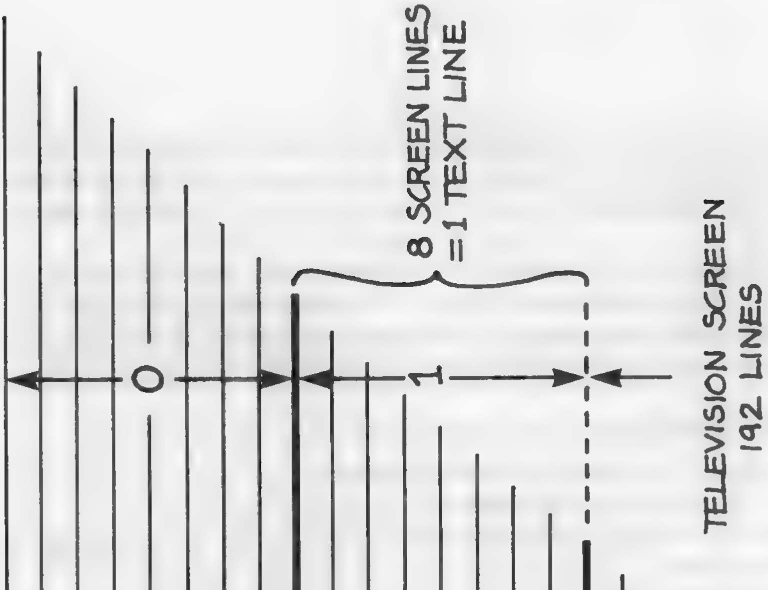
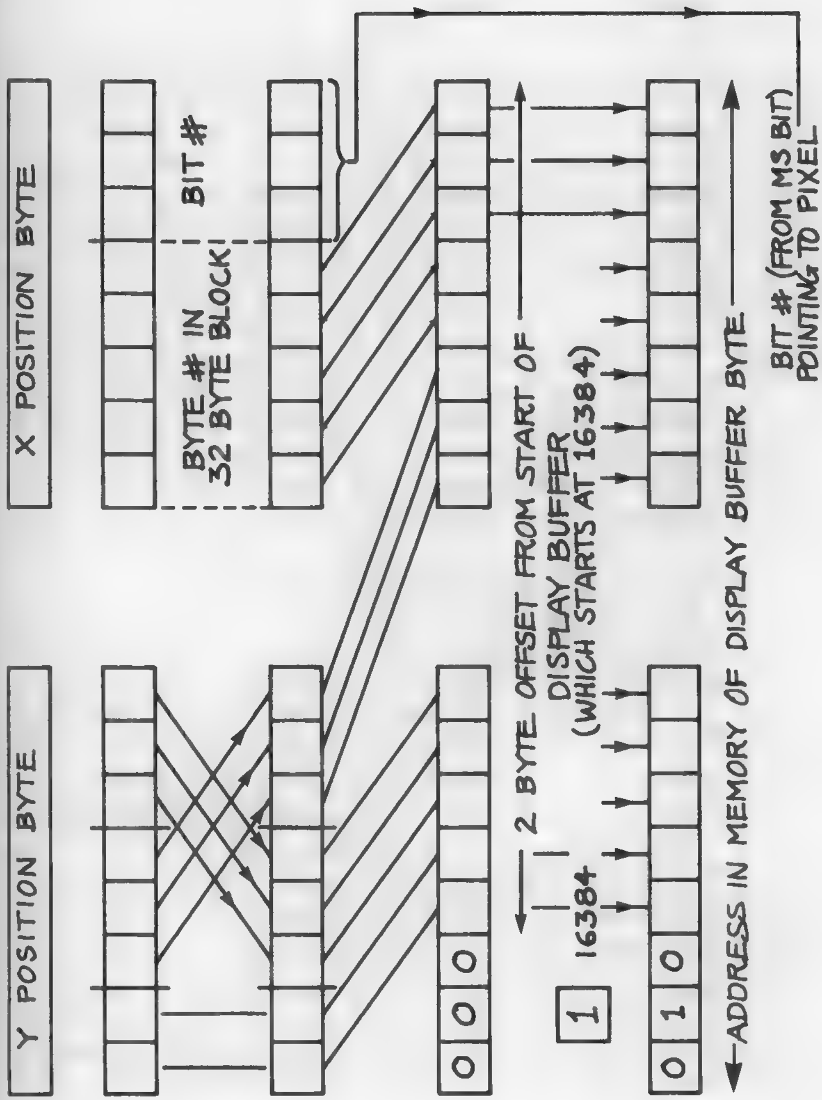

8 SCREEN LINES
=1 TEXT LINE
TELEVISION SCREEN
192 LINES
192
DISPLAY MEMORY
32 BYTE BLOCKS

Figure 7.1b


X POSITION BYTE
Y POSITION BYTE
BIT #
BYTE # IN 32 BYTE BLOCK!
0 0 0
1
16384
2 BYTE OFFSET FROM START OF DISPLAY BUFFER (WHICH STARTS AT 16384)
0 1 0
ADDRESS IN MEMORY OF DISPLAY BUFFER BYTE
BIT # (FROM MS BIT) POINTING TO PIXEL

Listing 7.1

```
a   PLOT  PUSH  BC
            PUSH  DE

            LD    A,C
b           AND   7
            ADD   1
            LD    E,A

            SRL   C
c           SRL   C
            SRL   C

            LD    A,B
            AND   56
d           SLA   A
            SLA   A
            OR    C
            LD    C,A

            LD    A,B
            AND   7
            LD    D,A
            LD    A,B
e           AND   192
            SRL   A
            SRL   A
            SRL   A
            ADD   D

f           ADD   64
            LD    B,A

g           PUSH  BC
            POP   HL

            LD    B,E
            LD    A,128
h           JR    PLA
      PLB   SRL   A
      PLA   DJNZ  PLB

            POP   DE
j           POP   BC
            RET
```

Write down the mapping of **Figure 7.1a** in octal:

Row 00 uses block 00
Row 01 uses block 10
Row 02 uses block 20
Row 10 uses block 01
Row 11 uses block 11
Row 20 uses block 02
Row 21 uses block 12

and light dawns!

For the 64 rows of each section, all we have to do is swap the two least significant octal digits of the row number (which is the y coordinate) to get the 32 byte block number of the section. The remaining two bits of the row number must then be 00, 01 or 10 to select which of the three sections we want. (11 is an illegal value.)

Now that we know what we want to do we can draw (**Figure 7.1b**) a bit manipulation diagram. From here the coding is more or less straightforward, but note that it is all done in registers. Where a routine is to be used frequently memory access operations are to be avoided as they take half as long again as register access operations. Later on, in writing characters, this routine will be called eight times per character or 6144 times for a full screen.

Now for the formalities and the program description:

Routine PLOT

Entry Conditions	x position in C register y position in B register
Exit Conditions	BC as at entry DE as at entry HL address of display buffer byte A one set bit corresponds to the bit in the byte addressed by HL which refers to the BC defined pixel

Note

1 The target bit in the display buffer is only indicated.

2 Since the program is 'drop through' (except for the A register shift), there is no flow diagram.

The above is an example of the documentation I mentioned previously. Now for the program description.

SECTION	DESCRIPTION
a	Save registers.
b	Mask out the bit number bits, add 1 and save the count in the E register (see (h) below for the reason for this addition).
c	Shift the contents of the C register three bits right (this forms the index within a 32 byte block).
d	Part 1 of the octal swap; (56 decimal = 70 octal) move the contents of B into A, mask with 56, move two places left and place these three bits in the MS 3 bits of the C register. (Along with the five bits which point to the byte within the block.)
e	Part 2 of the octal swap; extract the LS 3 bits from the B register and store them in D. 192 decimal is 128 + 64 or the MS 2 bits of a byte, extract these bits from the B register (they point to which section is needed), move them right three places and add them to the 3 bits in the D register.
f	The display buffer starts at 16384, which is bit 6 (decimal 64) in the MS byte of a 2 byte address; add 64 to the total in the A register and store the result in B.

Note: BC is now set up with the required address (on the assumption that BC pointed to a valid pixel to start with). There remains the problem of setting up the A register.

g	BC is transferred to HL.
h	B is set to the contents of E from stage b. This is one more than the count in the LS 3 bits because the decrement of B by the DJNZ operation is done before the right shift. Bit 7 is set in A and the PLB / PLA instruction pair shift A right as long as B is non zero. Exit is with A having one bit set in the correct place.
j	Restore BC and DE; A and HL are set up as required.

Exit

This is probably the most complicated routine in the whole book. Everything which outputs to the display uses it and unless you understand exactly how it works other things later on will probably be more difficult.

The Spectrum system allows the BASIC user to position the head of a piece of text by using AT and takes a new line with the start of each new PRINT statement. In the next part of the program, where we output characters in various forms, the top lefthand corner of each 8 × 8 character pixel array is located on the screen by two 1 byte variables, LINE and COLM. Their relative positions must not be altered as they are used together to set up BC for a call on PLOT to determine which display buffer bytes are to be loaded.

To simplify matters, COLM is incremented by 8 and when it overflows and becomes zero LINE is incremented by 8. When LINE points off the screen it is set to zero and display begins again at the top lefthand corner of the screen. The routine NPAGE sets both to zero and calls the display buffer clear routine CLRD.

Listing 7.2(1)

```
PRIN    PUSH AF
        PUSH BC
        PUSH DE
        PUSH HL
        SUB  32
        JP   M,PXL
        SUB  96
        JP   P,PXL
        ADD  96
        PUSH AF
        LD   BC,(COLM)
        CALL PLOT
        EX   DE,HL
        POP  HL
        LD   L,H
        LD   H,0
        ADD  HL,HL
        ADD  HL,HL
        ADD  HL,HL
        LD   BC,15616
        ADD  HL,BC
        LD   B,8
RPRT    LD   A,(HL)
        INC  HL
        EX   DE,HL
        LD   (HL),A
        INC  H
        EX   DE,HL
        DJNZ RPRT
        LD   A,(COLM)
        ADD  8
        LD   (COLM),A
```

```
1135            JR   NZ,PXL
1140            LD   A,(LINE)
1145            ADD  8
1150            LD   (LINE),A
1155            ADD  64
1160            JR   NZ,PXL
1165            LD   A,0
1170            LD   (COLM),A
1175            LD   (LINE),A
1180 PXL        POP  HL
1185            POP  DE
1190            POP  BC
1195            POP  AF
1200            RET
1205 COLM       NOP
1210 LINE       NOP
```

Listing 7.2(2)

```
1395 NPAGE      CALL CLRD
1400            PUSH AF
1405            LD   A,0
1410            LD   (LINE),A
1415            LD   (COLM),A
1420            POP  AF
1425            RET
```

PRIN

To display a single character at the location defined by LINE and COLM; also to set LINE and COLM to point to the next character position.

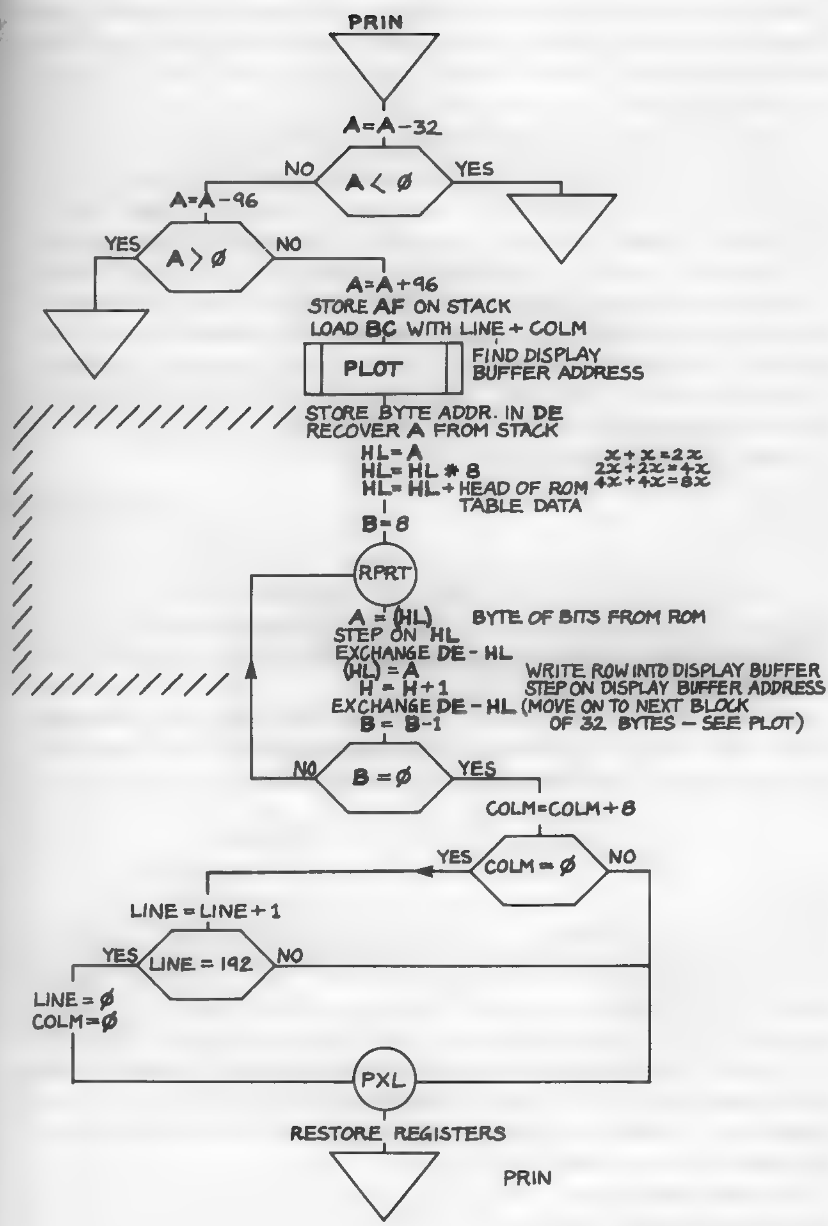
This routine uses the Spectrum ROM character table of pixel bit patterns, at 8 bytes per character for all the ASCII codes from 32 to 127 inclusive. They start at 15616 in ROM and each 8 byte block is set up along the lines indicated in Chapter 14 of the Spectrum manual.

The requirement in printing a character is to load into the display store the appropriate eight bytes and advance the LINE / COLM pointer(s) to be ready for the next character.

Commentary

This is acutely dependent on the LINE/COLM pointer indicating a character cell not crossing byte or display segment boundaries. I see no reason for complicating the problem, but see Chapter 8 for how to deal with the general problem.

Flowchart 7.1



The routine is entered with the ASCII character code in the A register and it is first tested for 'printability'. Non-printable characters are omitted, not replaced by blanks. 32 is effectively subtracted from the ASCII code, to give a position pointer to the bytes in the ROM, and A stored on the stack. PLOT is now used to determine the address of the display buffer byte to be used for the first row of pixels. LINE and COLM, stored as adjacent bytes, are collected together by the LD BC,.. operation.

From PLOT the byte address is stored in DE and the character code recovered, multiplied by 8 (8 bytes per character) and added to the head address of the ROM data table to point to the required bytes. This is the address to change if you want to use your own character definition bytes.

B is set to 8 to count the 8 bytes to be transferred from the ROM. That byte is transferred to the display buffer and the display buffer pointer incremented by 256 to point to the 32 byte block where the next byte is to be placed. This is done by incrementing the H register of the HL pair when it contains the appropriate data. The transfer loop at RPRT keeps its two pointers in HL and DE, exchanging them as needed. It starts off with HL pointing to the ROM and DE to the display buffer.

After the character has been written to the buffer COLM is incremented by 8 (8 pixels maketh one character row) to point to the next character in the line. If the count has gone over the top and become 0 LINE is incremented by 8 (8 pixel rows maketh one character) and the result tested against 192 for 'beyond bottom of screen'. If at the bottom both LINE and COLM are reset to zero. The routine exits after restoring registers, except the A, at PXL.

This routine will not work properly if either LINE or COLM come to contain any value which is not an exact multiple of 8. A first exercise for you is to modify it to ensure that they do stay as exact multiples of 8.

PTEX

Printing text, which is just a question of feeding PRIN with a sequence of characters, is achieved by arranging the text to be printed in the bytes immediately following the call of the routine and having it terminated with a zero byte. This works perfectly well with fixed length text, composed or allocated at assembly time, but you will need to produce a modified version which deals with text held somewhere else at a known address. I strongly advise you, however, to mark the end of such text with a zero byte as it is non-printable and easily tested for.

We saw in IFKEY how to use the subroutine return address from the top of the stack to get at data immediately following the call of a subroutine. We do the same thing here to get at the first, and subsequent, characters of the string. At PXB, with HL pointing to a byte, it is loaded into the A

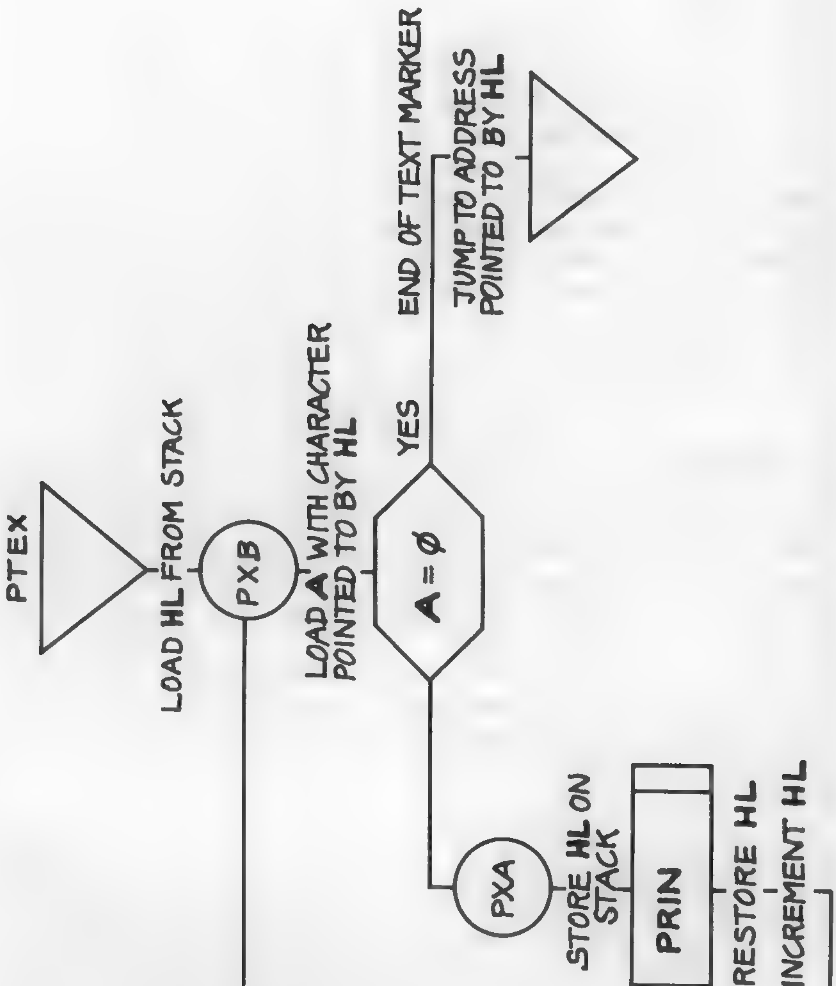
Listing 7.3(1)

```
1280 PTEX    POP  HL
1285 PXB     LD   A,(HL)
1290         CP   0
1295         JR   NZ,PXA
1300         JP   (HL)
1305 PXA     PUSH HL
1310         LD   A,(HL)
1315         CALL PRIN
1320         POP  HL
1325         INC  HL
1330         JR   PXB
```

Listing 7.3(2)

```
0700 SRHL$   SRL  L
0705         SRL  H
0710         RET  NC
0715         SET  7,L
0720         RET
0725 RHL3$   CALL SRHL$
0730         CALL SRHL$
0735         CALL SRHL$
0740         RET
```

Flowchart 7.2



register and compared with zero; if it is the end of text marker byte then HL is pointing to a NOP instruction as well and the JP (HL) instruction transfers control out of the routine and back to the main program; otherwise, at PXA, A contains an ASCII character to be displayed by PRIN. While PRIN runs the text pointer is saved on the stack so that it can be recovered and incremented. Return is then made to PXB to collect the next character or the end marker.

Now come two primitive routines for doing a double byte, 16 bit, right shift. There is a much better, more elegant and faster way of shifting right.

SRHL

The two SRL operations shift each register right 1 bit place, the second, on the more significant byte, will set the carry flag if a bit is 'lost' on the bottom and unset the flag if no bit was lost. The RET NC exits from the routine when no correction has to be made to the L register, otherwise the lost bit is replaced in the MS bit of the L register by the SET 7,L operation.

RHL3

This performs three right shift operations together, so dividing the contents of the HL register pair by 8 which is just what is required when printing octal numbers as described in PRT8.

PRT8

Now we can print text, what about numbers? Well, there are all sorts of complicated routines that you can read about elsewhere. This will just print a 16 bit binary number, in HL, as a 6 digit octal number, no frills, no sign, just something simple so that we can have a method of debugging programs later on.

There are two tricks here:

1 The ASCII characters for digits are a sequential set from 48 (decimal) onwards, so the required octal character is obtained by adding 48 to the 3 binary bits of the octal value in question.

2 Use the ready-built PTEX routine to do the output and overwrite what was output last time.

On entry the registers are all saved on the stack and DE pointed to the byte where the LS digit of the output is to be loaded for printing by PTEX. B is set to 6 as there are only six bytes to be produced. At PRU3 the three least significant bits of HL are obtained by masking and the character code for the value calculated by the addition of 48. This is stored in the location indicated by DE, DE is decremented and HL shifted right three bits to reveal the next octal group if B does not go to zero. If B is zero all six bytes

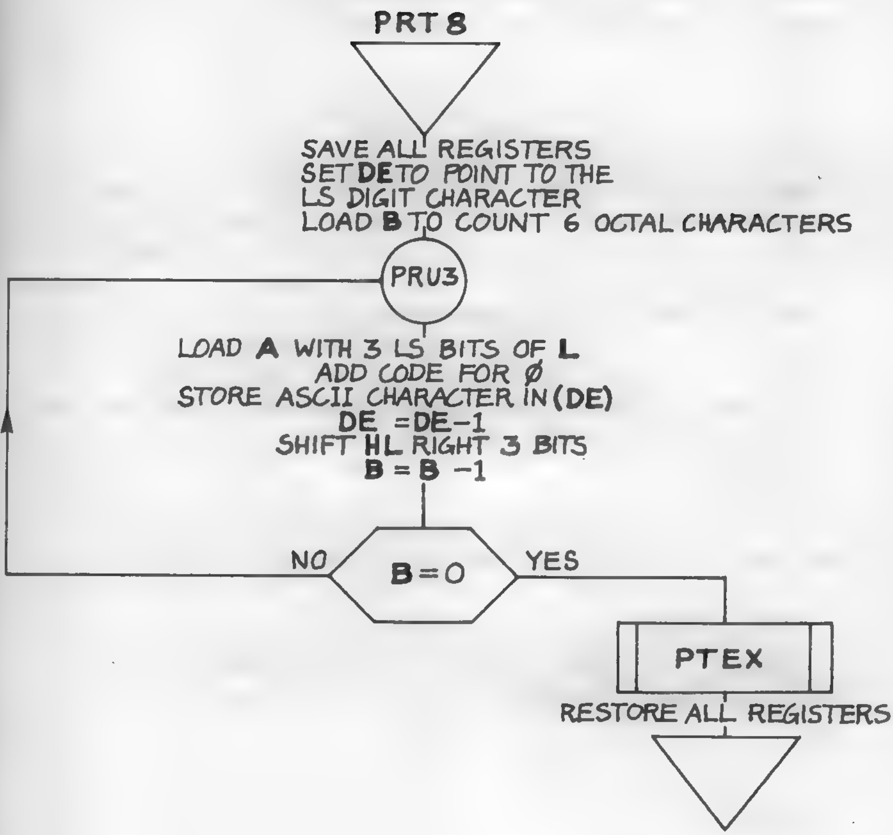
Listing 7.4(1)

```
1430 PRT8    PUSH AF
1435         PUSH BC
1440         PUSH DE
1445         PUSH HL
1450         LD   DE,P8Z1-1
1455         LD   B,6
1460 PRU3    LD   A,L
1465         AND  7
1470         ADD  48
1475         LD   (DE),A
1480         DEC  DE
1485         CALL RHL3$
1490         DJNZ PRU3
1495         CALL PTEX
1500         DEFM "cdefgh"
1505 P8Z1    DEFM "   "
1510         NOP
1515         POP  HL
1520         POP  DE
1525         POP  BC
1530         POP  AF
1535         RET
```

Listing 7.4(2)

```
5440 PRT8W   PUSH HL
5445         PUSH AF
5450         PUSH DE
5455         PUSH BC
5460         CALL PRT8
5465         CALL IFKEY
5470         DEFB "m"
5475         JP   PR8WX
5480         NOP
5485 PR8WX   POP  BC
5490         POP  DE
5495         POP  AF
5500         POP  HL
5505         RET
```

Flowchart 7.3



have been loaded on top of hgfedc in the listing, P8Z1 is a pair of spaces to terminate the displayed 6 characters output by the call of PTEX after which all the registers are restored to their entry values and the routine exits.

PRT8 can thus be inserted anywhere in a program when a check is required on the contents of HL.

RPORT

While debugging programs it is often necessary to be able to display the values of all the registers, so the next routine does exactly that, together with the return address using PRT8. Since the program is so straightforward there is no flow diagram.

There are three points to be noticed:

1 The stack pointer value, indicated by $= can indicate if the program is 'running away' because of unmatched POPs and PUSHs.

2 The CALL RPORT return address, indicated by #, allows several outputs from different calls to be distinguished.

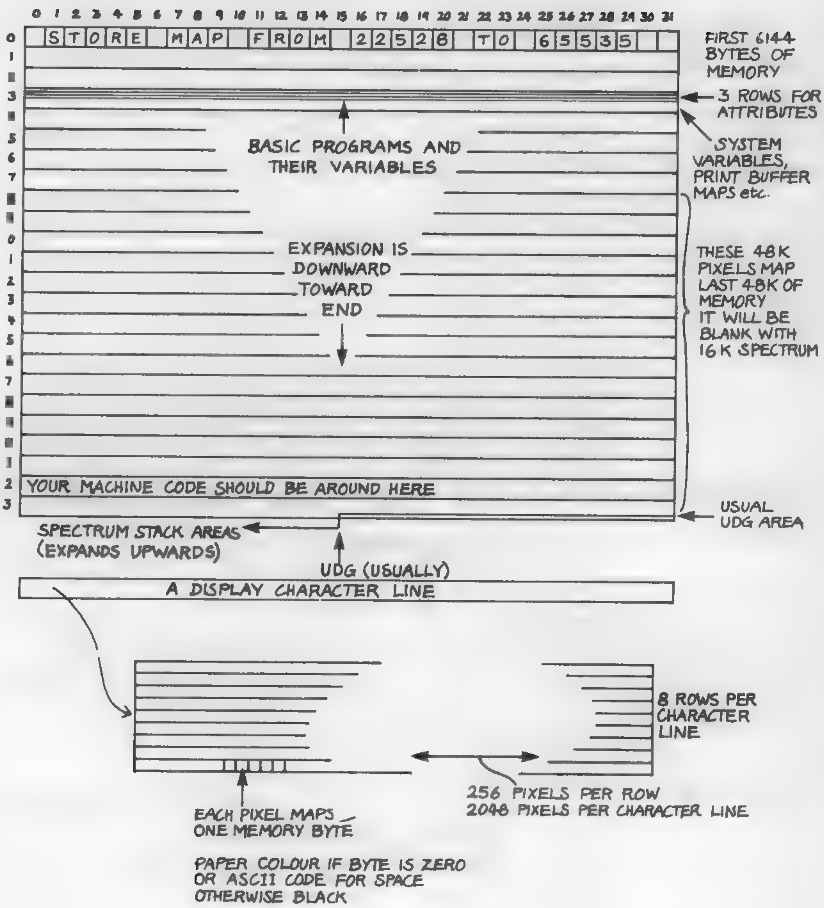
3 The messy way data is passed into HL to print the return address. There is a better, more elegant way by computing the instruction, as is demonstrated later, in MOVER and VAR$1 for example.

Listing 7.5

```
1540 RPORT  LD    (SP$),SP
1545        PUSH  AF
1550        PUSH  BC
1555        PUSH  DE
1560        PUSH  HL
1565        LD    (HL$),HL
1570        LD    (DE$),DE
1575        LD    (BC$),BC
1580        PUSH  AF
1585        POP   HL
1590        LD    (AF$),HL
1595        CALL  PTEX
```

```
            DEFM "AF="
            NOP
            LD   HL,(AF$)
            CALL PRT8
            CALL PTEX
            DEFM "BC="
            NOP
            LD   HL,(BC$)
            CALL PRT8
            CALL PTEX
            DEFM "HL="
            NOP
            LD   HL,(HL$)
            CALL PRT8
            CALL PTEX
            DEFM "DE="
            NOP
            LD   HL,(DE$)
            CALL PRT8
            CALL PTEX
            DEFM "$="
            NOP
            LD   HL,(SP$)
            CALL PRT8
            CALL PTEX
            DEFM "£="
            NOP
            LD   HL,(SP$)
            LD   E,(HL)
            INC  HL
            LD   D,(HL)
            EX   DE,HL
            CALL PRT8
            CALL PAUSE
            POP  HL
            POP  DE
            POP  BC
            POP  AF
            RET
AF$         DEFW 0
HL$         DEFW 0
DE$         DEFW 0
BC$         DEFW 0
SP$         DEFW 0
```

Figure 7.2



STORE MAP FROM 22528 TO 65535
FIRST 6144 BYTES OF MEMORY
3 ROWS FOR ATTRIBUTES
SYSTEM VARIABLES, PRINT BUFFER MAPS etc.
BASIC PROGRAMS AND THEIR VARIABLES
EXPANSION IS DOWNWARD TOWARD END
THESE 48K PIXELS MAP LAST 48K OF MEMORY IT WILL BE BLANK WITH 16K SPECTRUM
YOUR MACHINE CODE SHOULD BE AROUND HERE
USUAL UDG AREA
SPECTRUM STACK AREAS (EXPANDS UPWARDS)
UDG (USUALLY)
A DISPLAY CHARACTER LINE
8 ROWS PER CHARACTER LINE
256 PIXELS PER ROW
2048 PIXELS PER CHARACTER LINE
EACH PIXEL MAPS ONE MEMORY BYTE
PAPER COLOUR IF BYTE IS ZERO OR ASCII CODE FOR SPACE OTHERWISE BLACK

Map$

Now let us put some of the bricks together for something useful — a routine for displaying the free space in memory.

We have available 6144 bytes of display buffer which contain 48k of pixels so we can map each RAM byte to a pixel (we ignore the ROM) by setting the pixel to black ink if the byte is neither space, in ASCII, nor blank, otherwise the pixel is left paper coloured.

Commentary

The calls of NPAGE and PTEX clear the display and set up the output description in the top two lines; the top three text lines, or 24 rows of pixels map to the display buffer and we know what is there so I don't map that either.

From labels NXF1 to NXF3 the routine is setting up the attributes area so that each line of 8 rows of pixels is a different colour from its neighbours. Each row, remember, covers 8*256 = 2048 bytes of RAM and even rough location is impossible if the screen is all the same colour.

At NXF3 HL is set to the address of the head of RAM and BC, initialised to point to the first address beyond the end of the display buffer, and results in the sum HL + BC being the address of the byte to be currently tested. If this sum runs beyond 16 bits to zero the carry flag will be set and the routine exits on the RET C after all the RAM has been examined. The byte addressed by the sum, in HL, is loaded into the A register and tested for 0/32, in either case of equality the PLOT routine is skipped and the pixel left as paper colour.

Since the byte count is from the head of RAM in BC the lower byte can specify a pixel x position and the higher byte the pixel row. It is of course the purest happenstance that this is the way that PLOT requires its input to be specified.

Setting the ink pixel is just ORing in the bit in the A register, after the call of PLOT, with the address specified by HL. The program then returns to NXF with an increment BC to test the next byte.

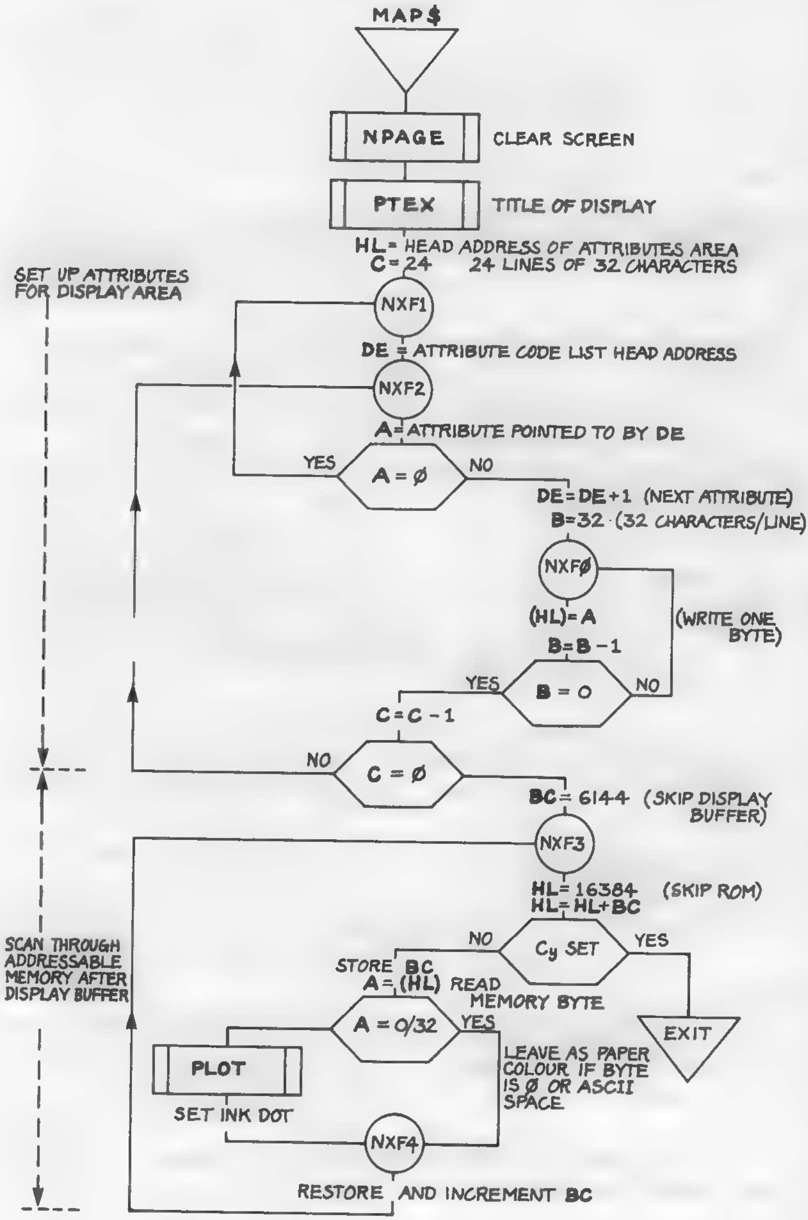
Synopsis

PLOT performs the same function as the Spectrum plot function, it is the foundation of all display output. It forms the basis of the animation routine of Chapter 8 and the drawing program of Chapter 13.

PRIN displays a character, ASCII code in the A register, at the next available character position.

NPAGE clears the screen and sets PRIN to start at the beginning of line one.

Flowchart 7.4



Listing 7.6

```
1820 MAP$     CALL NPAGE
1825          CALL PTEX
1830          DEFM " STORE MAP FROM 22528 TO
              65535  "
1835          NOP
1840          LD   HL,22528
1845          LD   C,24
1850 NXF1     LD   DE,LIST
1855 NXF2     LD   A,(DE)
1860          CP   0
1865          JR   Z,NXF1
1870          INC  DE
1875          LD   B,32
1880 NXF0     LD   (HL),A
1885          INC  HL
1890          DJNZ NXF0
1895          DEC  C
1900          JR   NZ,NXF2
1905          LD   BC,6144
1910 NXF3     LD   HL,16384
1915          ADD  HL,BC
1920          RET  C
1925          PUSH BC
1930          LD   A,(HL)
1935          CP   0
1940          JR   Z,NXF4
1945          CP   32
1950          JR   Z,NXF4
1955          CALL PLOT
1960          LD   B,(HL)
1965          OR   B
1970          LD   (HL),A
1975 NXF4     POP  BC
1980          INC  BC
1985          JR   NXF3
1990 LIST     DEFB 96
1995          DEFB 104
2000          DEFB 112
2005          DEFB 120
2010          NOP
```

PTEX uses PRIN to display the text following its call (the text must end with a zero value byte).

PRT8 uses PRIN to display the contents of HL as an octal number.

PRT8W uses PRT8 with a wait for the "m" key to be pressed.

RPORT displays the contents of the registers (not IX and IY).

MAP$ displays memory occupation.

CHAPTER 8
Animation

GCELL

The aim and object of this routine is to display rapidly a sequence of images at a moveable point on the screen. These images or patterns are drawn in a box or cell. The larger the cell, of up to 2040 pixels, the longer the routine takes. The BASIC interface is just about as complex as can be handled without designing a fundamentally new, and more general, technique: see Chapter 9.

Interface

The user sets bytes 23675/6 (UDG) to the address of the first byte of a block of data, defined below, which the routine will use. There may be several such blocks, switching amongst them is done by changing the contents of UDG.

BYTE	DESCRIPTION
0	cell horizontal position x
1	cell vertical position y
2	control flags and next frame no. *
3	BCR no. of bits per cell row 1 – 255
4	BCC no. of bits per cell column 1 – 255
5	WPC no. of works per cell 1 – 255
6	frame sequence control bytes; the LS 4 bits of byte 2 to one of these 15 bytes; the LS 4 bits of this byte define which cell is
20	to be displayed
21	set to 0
22	first byte of cell 1 data. There are WPC bytes in this and the other cells
22 + WPC	first byte of cell 2 data
22 + 2.WPC	first byte of cell 3 data

and so on for as many cells, up to 15, as are needed.

Control flags and next frame no. — byte 2

BIT NO.	DESCRIPTION
7	MS bit: if set the routine exits doing nothing
6	set by the routine if any part of the currently displayed cell is outside the allowable display area. This bit should be monitored by the user program
5–4	not used
3–0	if zero the routine exits If non zero the contents are used to point to a frame sequence control byte which identifies the next cell to be displayed. (Add 5 to the value and the result is 6 to 20; this is the number of a byte, relative to the head of the block, which contains the cell identifier). The routine increments this pointer or resets it to 1 after the end of the sequence list is met, as recognised by a zero entry. This can always be overwritten by the user rewriting byte (UDG) + 2

Cell data

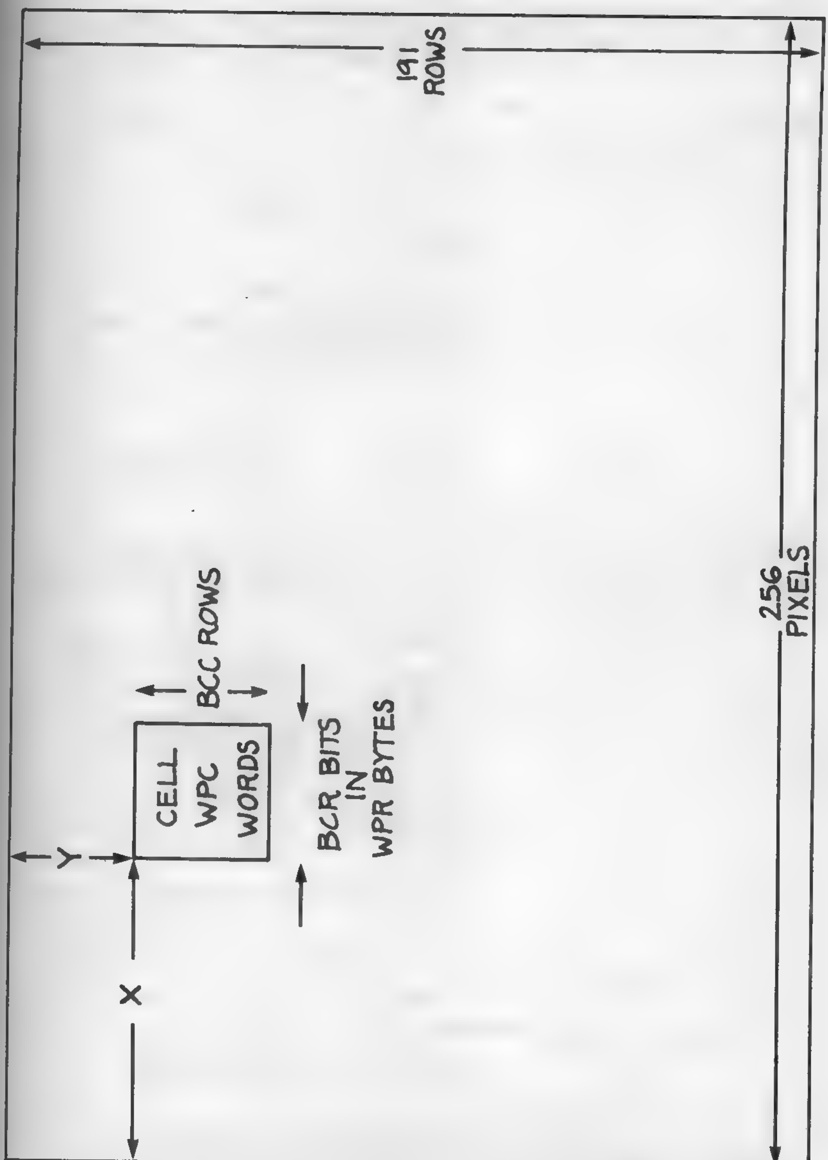
Each row of pixels in a graphics cell starts at the MS bit in the first of a sequence of bytes. There are BCR/8 bytes in this sequence, and surplus bits are ignored. There are BCC sequences, one for each row of pixels in the cell. Each set bit generates an inked pixel but remember that the attributes area must be set up as a separate exercise.

Description of GCELL

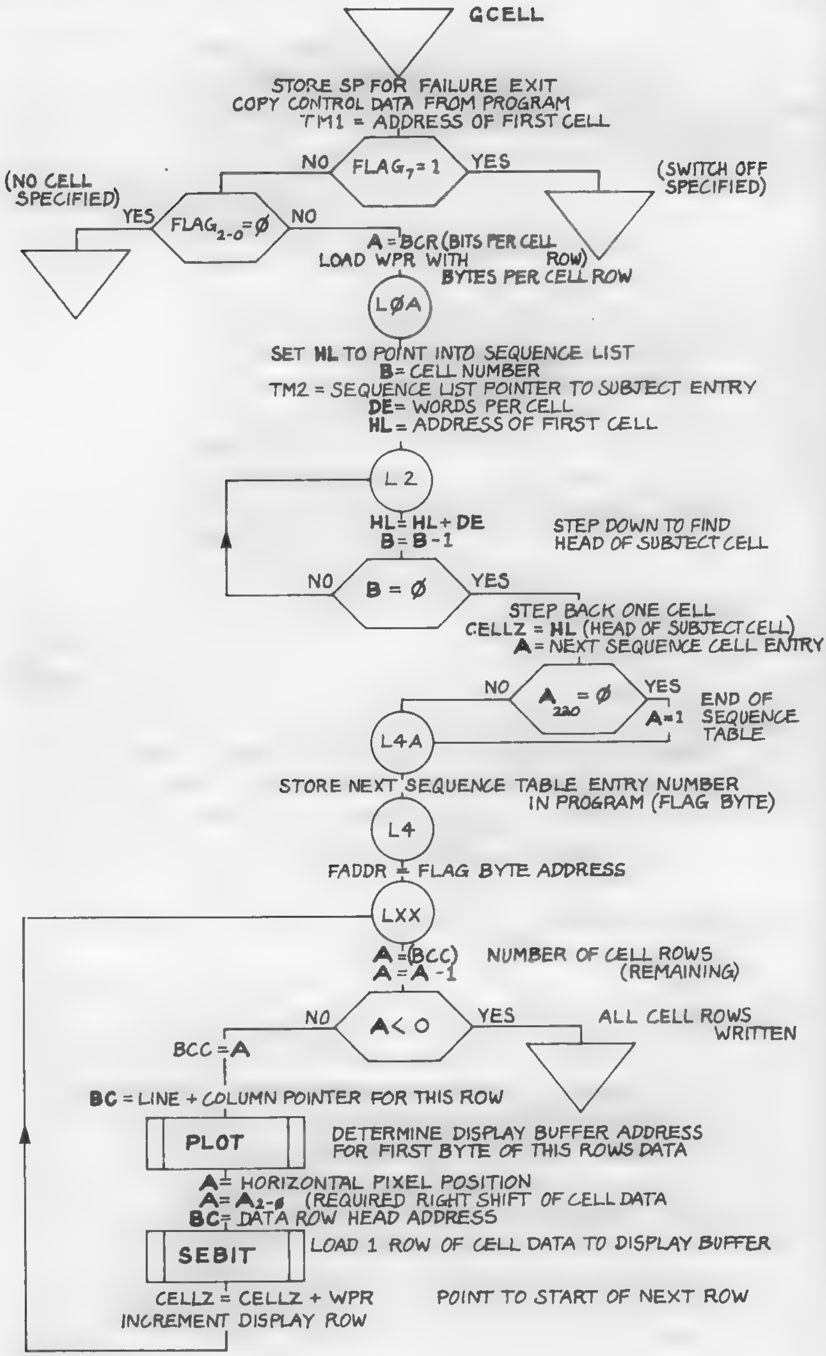
SP is stored in PANIC so the stack can be reset and a dignified exit made if the routine attempts to write beyond the allowable display buffer area. The first 21 bytes of the control data is copied into the routine and TM1 set to the address of the first graphics cell. Byte 2, the flag byte, is now tested and the routine exits if 'switch off' or no cell is specified, otherwise the last four bits specify which sequence table entry contains the required cell number. L0A to L2 and DJNZ operation pick up this cell number and set CELLZ with the head address of the cell data.

At L4A–L4 the next sequence table pointer is calculated and stored back in the interface table byte 2 ready for use at the next call of the routine. FADDR is set up with the address of this byte for use by SEBIT if need be.

Figure 8.1


191
ROWS
256
PIXELS
BCC ROWS
CELL
WPC
WORDS
BCR BITS
IN
WPR BYTES
X
Y

Flowchart 8.1



Listing 8.1

```
2015 GCELL  LD   (PANIC),SP
2020 UDG    EQU  23675
2025        LD   HL,(UDG)
2030        LD   DE,XY
2035        LD   BC,CELLZ-XY-1
2040        LDIR
2045        LD   (TM1),HL
2050        LD   A,(FLAG)
2055        BIT  7,A
2060        RET  NZ
2065        AND  15
2070        LD   (FLAG),A
2075        SUB  1
2080        RET  M
2085        LD   A,(BCR)
2090        SRL  A
2095        SRL  A
2100        SRL  A
2105        LD   (WPR),A
2110        LD   A,(BCR)
2115        AND  7
2120        JR   Z,LOA
2125        LD   A,(WPR)
2130        ADD  1
2135        LD   (WPR),A
2140 LOA    LD   HL,FSEQ-1
2145        LD   A,(FLAG)
2150        LD   B,0
2155        LD   C,A
2160        ADD  HL,BC
2165        LD   B,(HL)
2170        LD   (TM2),HL
2175        LD   A,(WPC)
2180        LD   E,A
2185        LD   D,0
2190        LD   HL,(TM1)
2195 L2     ADD  HL,DE
2200        DJNZ L2
2205        OR   A
2210        SBC  HL,DE
```

```
2215            LD    (CELLZ),HL
2220            LD    HL,(TM2)
2225            INC   HL
2230            LD    A,(HL)
2235            AND   15
2240            JR    NZ,L4A
2245            LD    A,1
2250 L4A        LD    HL,(UDG)
2255            INC   HL
2260            INC   HL
2265            LD    (HL),A
2270 L4         LD    (FADDR),HL
2275 LXX        LD    A,(BCC)
2280            SUB   1
2285            RET   M
2290            LD    (BCC),A
2295            LD    BC,(XY)
2300            CALL  PLOT
2305            LD    A,(XY)
2310            AND   7
2315            LD    BC,(CELLZ)
2320            CALL  SEBIT
2325            LD    HL,(CELLZ)
2330            LD    BC,(WPR)
2335            ADD   HL,BC
2340            LD    (CELLZ),HL
2345            LD    A,(XY+1)
2350            ADD   1
2355            LD    (XY+1),A
2360            JR    LXX
2365 PANIC      DEFW  0
2370 XY         DEFW  0
2375 FLAG       DEFB  0
2380 BCR        DEFB  0
2385 BCC        DEFB  0
2390 WPC        DEFB  0
2395 FSEQ       DEFM  "cD.N.Laine 1983"
2400            NOP
2405 CELLZ      DEFW  0
2410 WPR        DEFW  0
2415 TM1        DEFW  0
2420 TM2        DEFW  0
2425 FADDR      DEFW  0
```

Cell plotting (LXX onwards)

On entry XY holds the position of the top lefthand corner of the cell in PLOT required format, the other rows are defined by incrementing the y part of XY. BCC is used as a counter of the number of rows to be output and the routine exits when BCC has been decremented below zero.

PLOT determines the display buffer load byte address and bit number for the head of the current row of pixels which is pointed to by CELLZ, and SEBIT plots the row of pixels; CELLZ is then incremented to point to the head of the next row of pixels and the loop repeated.

SEBIT

This routine plots or unplots pixels in the display buffer according to how they are set or unset in the cell row. Separate pointers are maintained to step through both sets of bytes.

At entry to DNB HL points into the display buffer and DS points to the bit to be set/unset; CELLZ points into the cell data and TS is the bit number of the cell byte; WC is a count of bits/pixels per cell row — the routine loops BCR times.

TST is a computed bit test instruction to test the TS bit in the cell byte, according to how the bit is to be set/unset. So SET or RES instructions are computed and excuted at DO to set or unset the required bit in the display buffer.

Having dealt with one pixel the pointers are incremented; if either bit pointer is negative it is reset to 7 and the corresponding byte pointer is incremented. If the display buffer pointer, HL, points into the last pixel row then, since to simplify matters this is forbidden, the flag byte has bit 6 set and there is a PANIC escape.

Note: this uses computed instructions: how will your assembler deal with them?

64 + 7 generates the BIT ?,A instruction
128 + 7 generates the RES ?,A instruction
192 + 7 generates the SET ?,A instruction

Synopsis

GCELL, which uses PLOT, enables you to do complex animation. Chapter 13 will let you set up blocks of colour. Later on there is a routine to move the blocks around, or so it will appear.

Flowchart 8.2

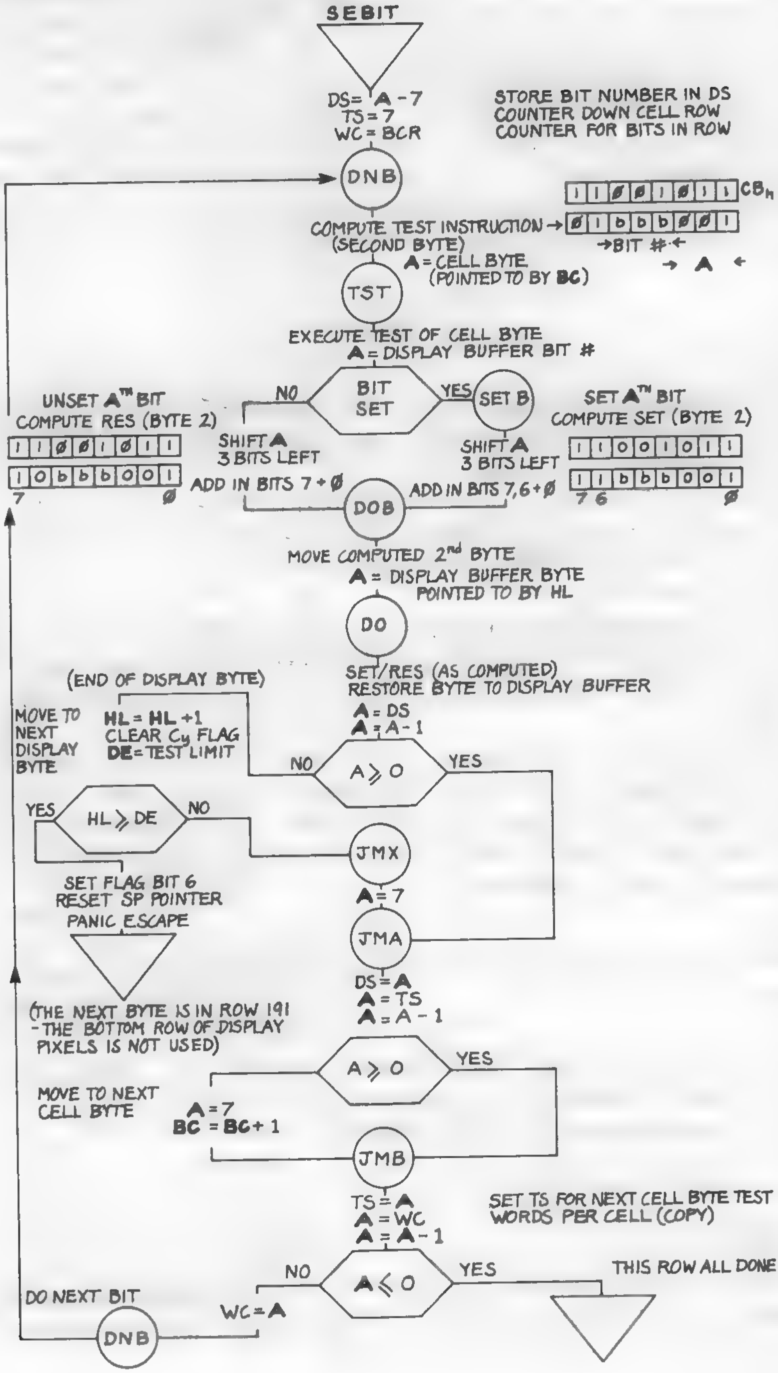
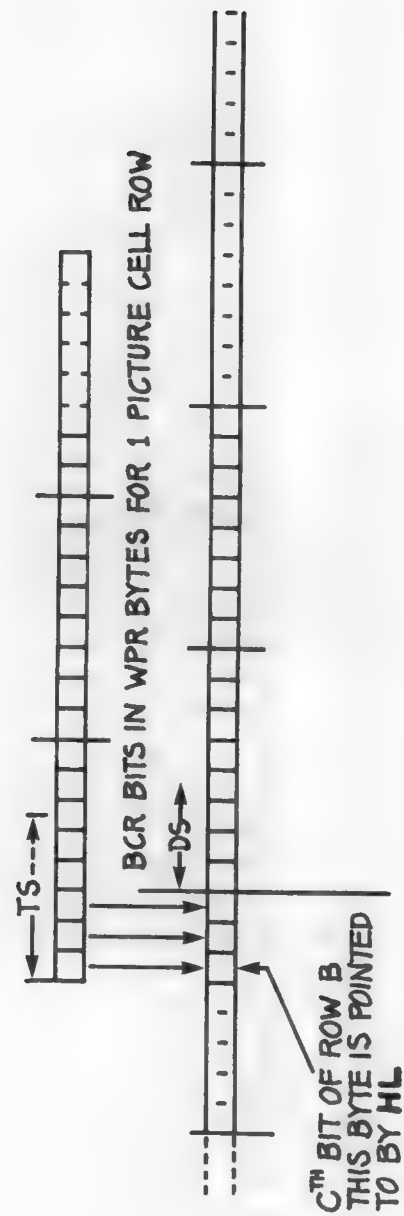


Figure 8.2


BCR BITS IN WPR BYTES FOR 1 PICTURE CELL ROW
TS
DS
CTH BIT OF ROW B
THIS BYTE IS POINTED
TO BY HL

Listing 8.2

```
2430 SEBIT   SUB   7
2435         NEG
2440         LD    (DS),A
2445         LD    A,7
2450         LD    (TS),A
2455         LD    A,(BCR)
2460         LD    (WC),A
2465 DNB     LD    A,(TS)
2470         SLA   A
2475         SLA   A
2480         SLA   A
2485         ADD   64+7
2490         LD    (TST+1),A
2495         LD    A,(BC)
2500 TST     BIT   1,A
2505         LD    A,(DS)
2510         JR    NZ,SETB
2515         SLA   A
2520         SLA   A
2525         SLA   A
2530         ADD   128+7
2535         JR    DOB
2540 SETB    LD    A,(DS)
2545         SLA   A
2550         SLA   A
2555         SLA   A
2560         ADD   192+7
2565 DOB     LD    (DO+1),A
2570         LD    A,(HL)
2575 DO      SET   0,A
2580         LD    (HL),A
2585         LD    A,(DS)
2590         DEC   A
2595         JP    P,JMA
2600         INC   HL
2605         OR    A
2610         LD    DE,22496
2615         SBC   HL,DE
2620         ADD   HL,DE
2625         JP    M,JMX
2630         LD    HL,(FADDR)
```

```
2635          LD    A,(HL)
2640          OR    64
2645          LD    (HL),A
2650          LD    HL,(PANIC)
2655          LD    SP,HL
2660          RET
2665 JMX      LD    A,7
2670 JMA      LD    (DS),A
2675          LD    A,(TS)
2680          DEC   A
2685          JP    P,JMB
2690          LD    A,7
2695          INC   BC
2700 JMB      LD    (TS),A
2705          LD    A,(WC)
2710          DEC   A
2715          RET   M
2720          RET   Z
2725          LD    (WC),A
2730          JR    DNB
2735 DS       DEFB  0
2740 TS       DEFB  0
2745 WC       DEFB  0
```

CHAPTER 9

Error Handling and Parameter Name Passing

Error return handling

The notes on the stack (Chapter 3) mention a method of escape from a piece of code if some insoluble or unforeseen condition is encountered (see also Chapter 8 and the use of PANIC). In such an event it is very useful to be able to output some indication of the problem.

Machine code seems nearly always to be called by RANDOMISE USR... There is no fundamental need to do this; Chapter 26 page 180 of the Spectrum manual uses PRINT USR 32500 to print the contents of the BC register (as set up by the machine code); if we code such that they always exit abnormally with BC O we can call them by:

```
IF USR...< > 0 THEN GOTO...error routine
```

or, better

```
LET errorcode = USR...
IF errorcode < > 0 THEN GOTO...
```

since we can design the non zero value to have some special significance.

All these IF... < > 0 THEN GOTO... are unslightly and (worse still) are in BASIC. Look at the variables NEWPPC and NSPPC in Chapter 25 page 174 of the Spectrum manual.

NSPPC tells us exactly what to do. We design the BASIC part of our program such that some line, say 2, is the line to be jumped to in an error condition. Our error exit must then contain:

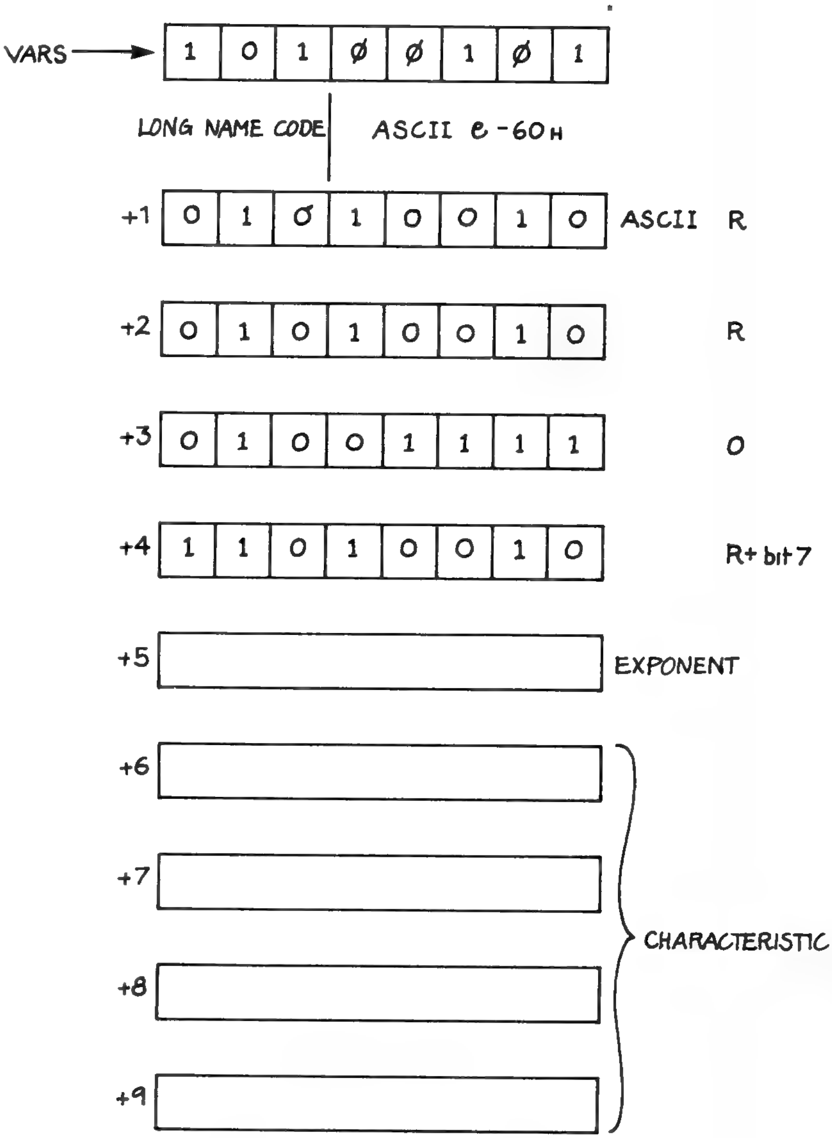
```
LD HL,2
LD (23618),HL
LD A,1
LD (23620),A
```

which pokes NEWPCC with 2 and NSPCC with 1. Lo and behold, we arrive at line 2 in BASIC.

We now undertake to call our routine by:

```
LET errorcode = USR...
```

Figure 9.1



HEAD OF VARIABLES AREA

and continue normally with the next statement. With any error exit we arrive at line 2 and errorcode contains the value of BC when the exit was made. Strange as it seems, the assignment of BC to errorcode is made regardless of how the exit is made.

There is just one small snag on the horizon. We can use BC as a means of passing information out of the machine back into the BASIC world, and it would be a pity to waste this facility for error routines. After all, good programs like ours do not meet error conditions (!) Can we get the error information out some other way?

Again the Spectrum manual has the answers, well hidden away in the depths of Chapters 24 & 25. It is a somewhat roundabout path but I can promise some primroses by the way.

1 23627/8 — VARS contains the address of the head of the variables tables in the BASIC program.
2 Pages 166 – 170 shows how these variables and their names are organised.

If the FIRST executed line in the BASIC is:

```
0001 LET ERROR = 0 : GOTO...
```

then the head of the variables area will be as **Figure 9.1** and the error routine in the machine code program can locate bytes 5 – 9 and insert values as required into the variable 'error'. The key here is that the *first* statement in the program forces the *first* variable to be located at the head of the variables area. At any point in the running of a BASIC program the sequence of the variables in the variables area will probably depend on the way in which that point in the program was reached since entries are made, as required by LET ??? = . . . statements, as they are encountered.

Our program now looks something like:

```
0001  LET error = 0: GOTO 100
0002  PRINT "ERROR CONDITION = ";error
0003......error handling routines...
0004

0100  REM program proper starts here
0101
...
0150  LET q=USR...
```

and q is some useful value generated by the routine and passed back to the BASIC program via the BC register.

Note now:

1 BC only allows you to get a value from code into BASIC.
2 'Error' can, if you want, be a 5 byte Spectrum floating point number of a Spectrum integer value.
3 We can now communicate between code and one BASIC variable, and, by extension, the BASIC program.
4 We could even change the use of 'error' and use it as an input variable to the machine code.

We can stop here, or go on to develop a method of passing variable names and values (parameters) between the machine code and the Spectrum BASIC program. We need to be able to do two things:

1 Pass variable names to the routine
2 Given a variable name, we have to find its address in the variables area of the BASIC program.

Let us first formalise how we are going to deal with error conditions.

On program entry we undertake to ensure that the first BASIC variable has a five-character name which will be reserved for passing error codes. We further undertake that our error handling routine(s) will start at line 2.

The machine code routine(s) all commence by storing the current value of the stack pointer for the (exclusive) use of the escape mechanism.

The escape mechanism shall return to the BASIC program the then current contents of the DE register pair, into this first BASIC variable, and force the return to line 2.

Entry to the escape mechanism is with DE set to a suitable code value and a JP, CALL, or JR operation as seems appropriate.

The routine is listed in **Listing 9.1**. ERROR is the stack pointer reset value set up at the routine call.

Listing 9.1(1)

```
1215 ERREX   LD    HL,(ERROR)
1220         LD    SP,HL
1225         LD    HL,2
1230         LD    (23618),HL
1235         LD    A,1
1240         LD    (23620),A
1245         LD    HL,(23627)
1250         LD    BC,7
1255         ADD   HL,BC
```

```
1260          LD    (ERADR+2),HL
1265 ERADR    LD    (ERADR+2),DE
1270          RET
1275 ERROR    DEFW 0
```

Listing 9.1(2)

```
0005          ORG   60000
0010          LD    (ERROR),SP
0015          PUSH AF
0020          PUSH BC
0025          PUSH DE
0030          PUSH HL
0035          PUSH IX
0040          CALL DRAWL
0045          JP    TRAP$
0050          LD    (ERROR),SP
0055          PUSH AF
0060          PUSH BC
0065          PUSH DE
0070          PUSH HL
0075          PUSH IX
0080          CALL SATTR
0085          JP    TRAP$
0090          LD    (ERROR),SP
0095          PUSH AF
0100          PUSH BC
0105          PUSH DE
0110          PUSH HL
0115          PUSH IX
0120          CALL BLOCK
0125          JP    TRAP$
0130          LD    (ERROR),SP
0135          PUSH AF
0140          PUSH BC
0145          PUSH DE
0150          PUSH HL
0155          PUSH IX
0160          CALL SORTF
0165          JP    TRAP$
0170          LD    (ERROR),SP
0175          PUSH AF
```

```
0180            PUSH BC
0185            PUSH DE
0190            PUSH HL
0195            PUSH IX
0200            CALL GCELL
0205            JP   TRAP$
0210            LD   (ERROR),SP
0215            PUSH AF
0220            PUSH BC
0225            PUSH DE
0230            PUSH HL
0235            PUSH IX
0240            CALL MAP$
0245            JP   TRAP$
0250            LD   (ERROR),SP
0255            PUSH AF
0260            PUSH BC
0265            PUSH DE
0270            PUSH HL
0275            PUSH IX
0280            CALL IVERT
0285            JP   TRAP$
0290            LD   (ERROR),SP
0295            PUSH AF
0300            PUSH BC
0305            PUSH DE
0310            PUSH HL
0315            PUSH IX
0320            CALL MOVEC
0325            JP   TRAP$
0330            LD   (ERROR),SP
0335            PUSH AF
0340            PUSH BC
0345            PUSH DE
0350            PUSH HL
0355            PUSH IX
0360            CALL SVERT
0365            JP   TRAP$
0370            LD   (ERROR),SP
0375            PUSH AF
0380            PUSH BC
0385            PUSH DE
0390            PUSH HL
0395            PUSH IX
```

```
        CALL DRAWA
        JP   TRAP$
        LD   (ERROR),SP
        PUSH AF
        PUSH BC
        PUSH DE
        PUSH HL
        PUSH IX
        CALL DEMO1
        JP   TRAP$
        LD   (ERROR),SP
        PUSH AF
        PUSH BC
        PUSH DE
        PUSH HL
        PUSH IX
        CALL DEMO2
        JP   TRAP$
TRAP$   POP  IX
TRAQ$   POP  HL
TRAR$   POP  DE
TRAS$   POP  BC
TRAT$   POP  AF
        RET
```

Passing variable names (parameters)

Chapter 25 page 174 of the Spectrum manual holds an answer to the problem. NXTLIN (location 23637/8) contains the head address of the next line of the BASIC program ie the one after the one which contains the LET... = USR... statement. We might put a list of parameters in this next line, hidden from the BASIC system by a REM statement.

A code call with parameters would then look like:

```
0175 LET y= USR 12345
0176 REM a, b : REM a and b are parameters of USR 12345.
```

Chapter 24 page 166 tells us how to get at the names. NXTLIN points to the MS byte of the line number so (NXTLIN) + 4 is the address of the first character of the text of that line; we step down the line looking for the REM token (= 234) and then we start looking for the variable name which we will specify to end in a comma, colon or an ENTER token. Spaces will be ignored and integers will be detected as they commence with a digit...

STOP !!!

It is very easy to get carried away when designing a piece of program; the specification becomes bigger and better, all singing and dancing, and much harder to debug; so much so that the program, which started as a good idea, becomes a bilious nightmare and is eventually abandoned in a mixture of disgust and despair.

Let us abandon, for the time being, passing numerical values and multicharacter variable names, and restrict ourselves to passing a limited number of single letter variables (which may be simple variables, strings or arrays). We can always go and complicate matters later on.

Flowchart 9.1 is a reproduction of the original flow diagram; **Flowchart 9.2** is the final version which ties up with **Listing 9.2**. The box VAR$1 is another routine which searches the variables area looking for names (**Flowchart 9.3**). It returns either A=0 end of data, or HL holding the address of the head of the variable name and A = first 3 (code) bits of that name: see below for the details.

The routine does not do exactly what it might have been thought it would do. The letters, brackets and $ can be in any order and the effective variable identifier is the last letter. The REM statement must be terminated by a colon or ENTER token. It is left to you, the reader, as a simple exercise to remedy these defects if you want to.

Documentation

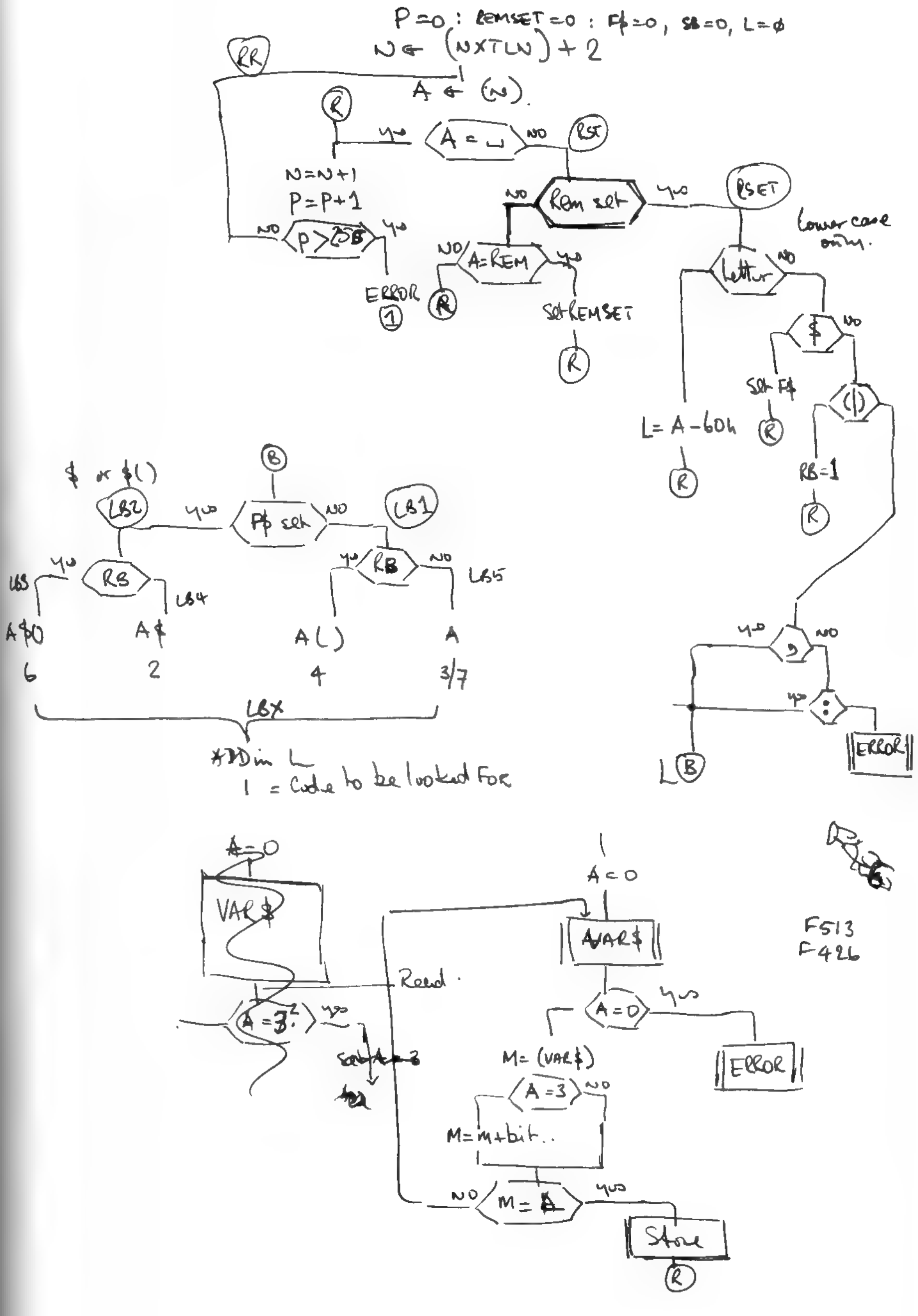
Entry conditions	none
Exit conditions	All registers lost PARM0 – PARM6 are the addresses of the first characters of up to seven variable names in the BASIC variables area. 0 indicates that no parameter is present

Note: The calling routine must verify that the type of the received variable is correct and extract/load the appropriate bytes.

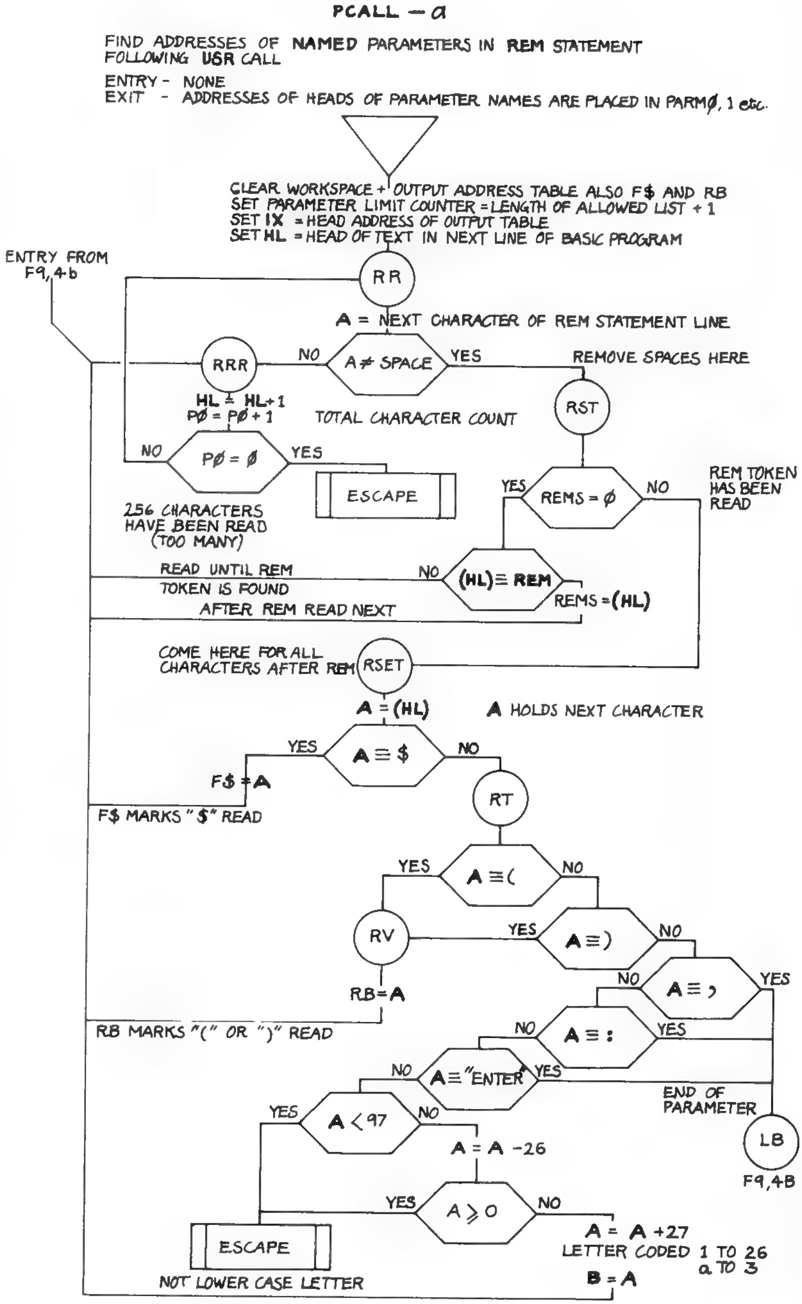
PCALL

The number of parameters to be handled, PARM0, PARM1,... is calculated at assembly time and PEND, their count + 1, is set up on entry. The right shift allows for two bytes of storage for each PARM location. For more parameters just add 2 byte storage as required between PARM6 and PEND; the LDIR operation will clear everything on entry. Remember, the PARM list contains the ADDRESSES of the head of each variable name.

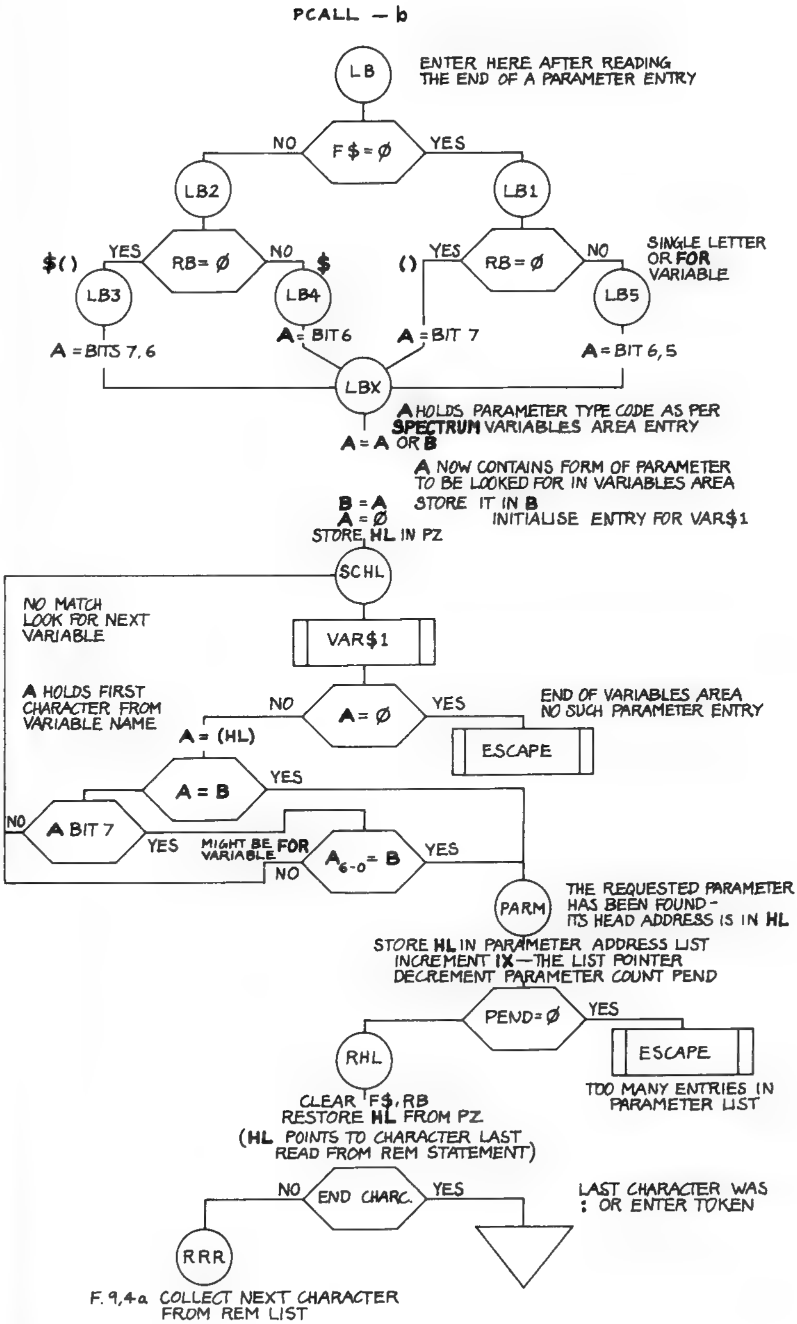
Flowchart 9.1



Flowchart 9.2a



Flowchart 9.2b



Listing 9.2

```
PCALL   LD    HL,0
NXTLN   EQU   23637
        LD    (PO),HL
        LD    HL,PO
        LD    DE,PO+1
        LD    BC,PEND-PO
        LDIR
        LD    A,PEND+3-PARMO
        SRL   A
        LD    (PEND),A
        LD    IX,PARMO
        LD    BC,4
        LD    HL,(NXTLN)
        ADD   HL,BC
RR      LD    A,(HL)
        CP    " "
        JR    NZ,RST
RRR     INC   HL
        PUSH  HL
        LD    HL,PO
        INC   (HL)
        POP   HL
        JR    NZ,RR
        LD    DE,4
        CALL  ERREX
RST     LD    A,(REMS)
        CP    0
        JR    NZ,RSET
        LD    A,(HL)
        CP    234
        JR    NZ,RRR
        LD    (REMS),A
        JR    RRR
RSET    LD    A,(HL)
        CP    "$"
        JR    NZ,RT
        LD    (F$),A
        JR    RRR
RT      CP    "("
        JR    NZ,RU
RV      LD    (RB),A
```

```
2955          JR    RRR
2960 RU       CP    ")"
2965          JR    Z,RV
2970          CP    ","
2975          JR    Z,LB
2980          CP    ":"
2985          JR    Z,LB
2990          CP    13
2995          JR    Z,LB
3000          SUB   97
3005          JP    M,ERX2
3010          SUB   26
3015          JP    P,ERX2
3020          ADD   27
3025          LD    B,A
3030          JR    RRR
3035 LB       LD    A,(F$)
3040          CP    0
3045          JR    Z,LB1
3050 LB2      LD    A,(RB)
3055          CP    0
3060          JR    Z,LB4
3065 LB3      LD    A,128+64
3070          JR    LBX
3075 LB4      LD    A,64
3080          JR    LBX
3085 LB1      LD    A,(RB)
3090          CP    0
3095          JR    Z,LB5
3100          LD    A,128
3105          JR    LBX
3110 LB5      LD    A,64+32
3115 LBX      OR    B
3120          LD    B,A
3125          LD    A,0
3130          LD    (PZ),HL
3135 SCHL     CALL  VAR$1
3140          CP    0
3145          JP    Z,ERX2
3150          LD    A,(HL)
3155          CP    B
3160          JR    Z,PARM
3165          BIT   7,A
3170          JR    Z,SCHL
```

```
3175          AND  127
3180          CP   B
3185          JR   Z,PARM
3190          LD   A,1
3195          JR   SCHL
3200 PARM     LD   (IX+0),L
3205          LD   (IX+1),H
3210          INC  IX
3215          INC  IX
3220          LD   A,(PEND)
3225          DEC  A
3230          LD   (PEND),A
3235          JR   Z,ERX4
3240          LD   A,0
3245          LD   (F$),A
3250          LD   (RB),A
3255 RHL      LD   HL,(PZ)
3260          LD   A,(HL)
3265          CP   ":"
3270          RET  Z
3275          CP   13
3280          RET  Z
3285          JP   RRR
3290 PO       DEFB 0
3295 REMS     DEFB 0
3300 F$       DEFB 0
3305 RB       DEFB 0
3310 PZ       DEFW 0
3315 PARM0    DEFW 0
3320 PARM1    DEFW 0
3325 PARM2    DEFW 0
3330 PARM3    DEFW 0
3335 PARM4    DEFW 0
3340 PARM5    DEFW 0
3345 PARM6    DEFW 0
3350 PEND     DEFB 0
3355 ERX1     LD   DE,1
3360          CALL ERREX
3365 ERX2     LD   DE,2
3370          CALL ERREX
3375 ERX3     LD   DE,3
3380          CALL ERREX
3385 ERX4     LD   DE,4
3390          CALL ERREX
```

The RR–RST–RRR part of the routine reads down the next program line until the REM token is found at which point the variable REMS is set non zero, thereafter RST will branch to RSET where the next non space character from the parameter REM statement is tested. F$ is set if a $ (string indicator character) is read; RB is set if either (or) is read (array indicator characters), a comma, colon or ENTER token (13) forces a jump to LB and then the character is tested to be a lower case letter. If it is then it is stored, less 60 (hex) in B; any character failing the test causes an error escape which sets DE = 2 and calls ERREX.

When LB is reached a parameter name has been read, and the program LB to LBX examines F$, RB to determine the required type and form, with the identifying letter, the entry to be looked for in the variables area.

A is set to zero to initialise VAR$1 on its first call at SCHL. VAR$1 exits with A = 0 if no more variables exist and the routine escapes with error 2, otherwise, HL points to the head of a variable name which is then compared with the subject being searched for. If not yet found the program returns to SCHL with A non zero otherwise PARM makes the head of the code which stores the head address of the variable in the parameter list and checks, first, that there is room for it. The markers F$, RB are reset and the routine returns to RRR to read the next character from the parameter list or exits if the end of the list was met.

VAR$1

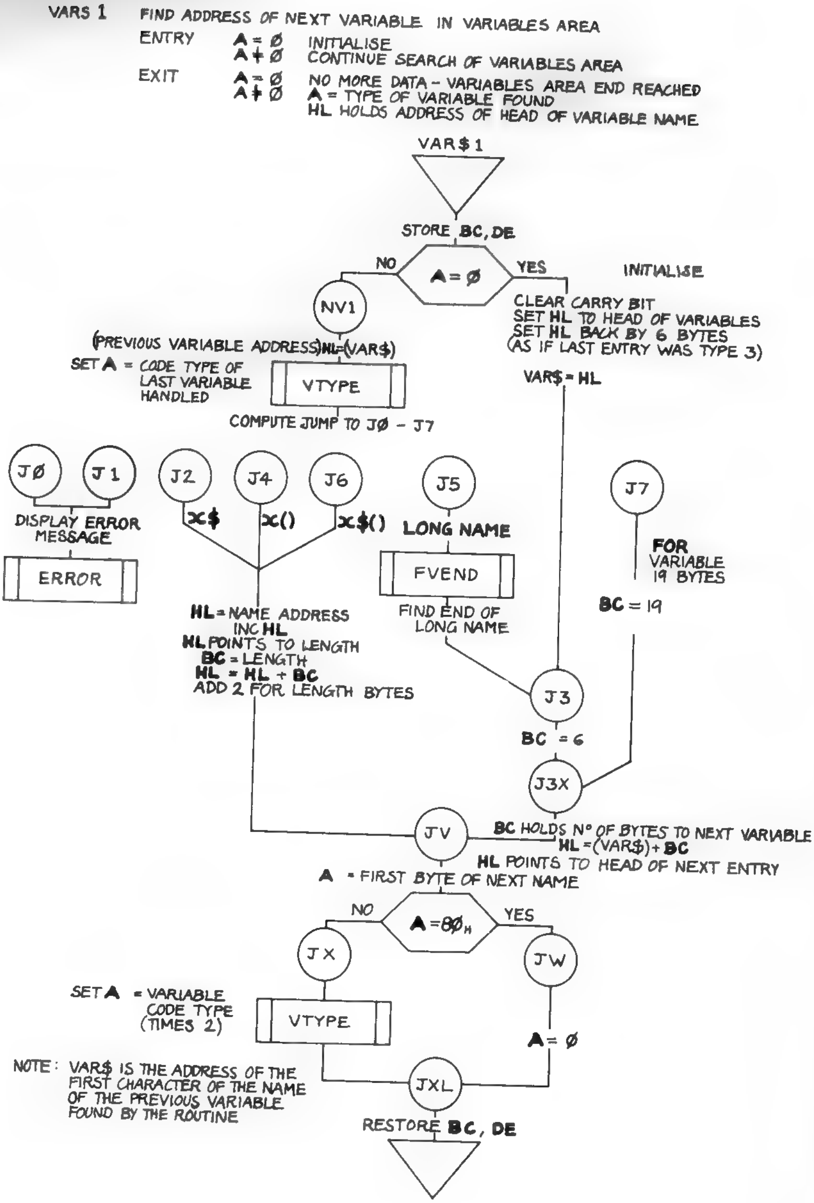
The variables area consists of a sequential table of names and data whose head address is given by the contents of 23627/8. Chapter 24 page 166–8 of the Spectrum manual defines the format and coding of all of them and VAR$1 uses this to deliver variable addresses.

The first three bits of each variable define its type and so enable the start of the next variable to be found. These bits are:

000	not used
001	not used
010	string
011	single letter name variable
100	array of numbers
101	multiple character name
110	array of characters
111	FOR loop control variable

These codes are used to compute a relative jump at JNV to another relative jump which deals with finding the end of this variable and the start of the next. The whole process is started off by thinking of the previous, non-existent variable, as being of type 3 by offsetting the head of the variables area by 6 and jumping to J3.

Flowchart 9.3



The labels J0 to J7 identify the sections of code which deal with the corresponding variable types as indicated in **Flowchart 9.3**. With type 5 the end of the name is indicated by the setting of the MS bit in the last byte, FVEND looks for this and the routine proceeds as if a single letter variable name had been read.

Between calls of VAR$1 the variable VAR$ holds the position reached so far in the scan. During the routine HL points into the variables area.

Should the routine start to generate errors, after having worked correctly, I would suspect that the Spectrum variable VAR$ had been corrupted, or the variables area had been overwritten.

Note on the computed jump at JNV:
The instruction JR J0 is a two byte operation

byte 0 = 24 decimal, 18 hex
byte 1 = offset

Since the whole table consists of such jumps the required offsets will be 0, 2, 4, 6, etc. VTYPE can only produce the eight values 0 (1) 7 shifted left one bit, ie 0 (2) 14 and the table covers all the possibilities. The 0th and first entries, both impossible, jump to the error routine.

Listing 9.3

```
3395 VARSS   EQU  23627
3400 VAR$1   PUSH BC
3405         PUSH DE
3410         CP   0
3415         JR   NZ,NV1
3420         OR   A
3425         LD   HL,(VARSS)
3430         LD   BC,6
3435         SBC  HL,BC
3440         LD   (VAR$),HL
3445         JR   J3
3450 NV1     CALL VTYPE
3455         LD   (JNV+1),A
3460         LD   DE,666
3465 JNV     JR   J0
3470         JR   J0
3475         JR   J0
3480         JR   J2
3485         JR   J3
3490         JR   J4
```

```
3495          JR    J5
3500          JR    J6
3505          JR    J7
3510 JO       CALL  PTEX
3515          DEFM  "VAR$1  jO error"
3520          NOP
3525          CALL  ERREX
3530 J2       INC   HL
3535          LD    (VP+2),HL
3540 VP       LD    BC,(VP+2)
3545          ADD   HL,BC
3550          INC   HL
3555          INC   HL
3560 JV       LD    (VAR$),HL
3565          LD    A,(HL)
3570          CP    080H
3575          JR    NZ,JX
3580 JW       LD    A,0
3585          JR    JXL
3590 JX       CALL  VTYPE
3595 JXL      POP   DE
3600          POP   BC
3605          RET
3610 J3       LD    BC,6
3615 J3X      ADD   HL,BC
3620          JR    JV
3625 J4       JR    J2
3630 J5       CALL  FVEND
3635          JR    J3
3640 J6       JR    J2
3645 J7       LD    BC,19
3650          JR    J3X
3655 VAR$     DEFW  0
3660 VTYPE    LD    A,(HL)
3665          AND   128+64+32
3670          RLC   A
3675          RLC   A
3680          RLC   A
3685          RLC   A
3690          RET
3695 FVEND    LD    HL,(VAR$)
3700          INC   HL
3705 FV1      BIT   7,(HL)
3710          JR    NZ,FV2
```

```
3715          INC   HL
3720          JR    FV1
3725 FV2      LD    (VAR$),HL
3730          RET
```

Synopsis

PCALL passes the addresses of BASIC variables into your code routines. This greatly eases the problem of data passing and most of the following chapter routines use this or a related subroutine OPARS.

The first BASIC variable is assigned the name ERROR and line 2 of the BASIC program is reserved for error routines.

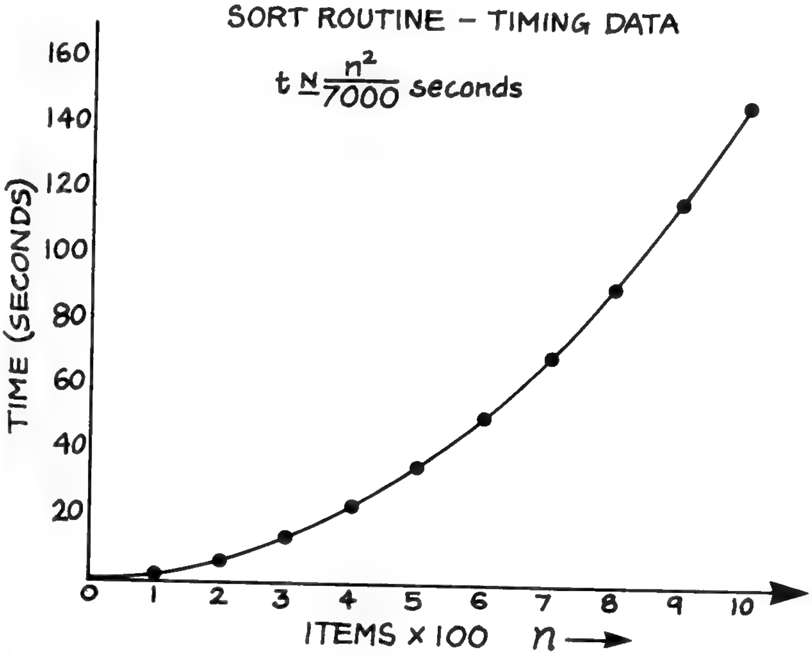
CHAPTER 10
Floating Point Array Sort

"'Beyond the mountains the grass is greener'"

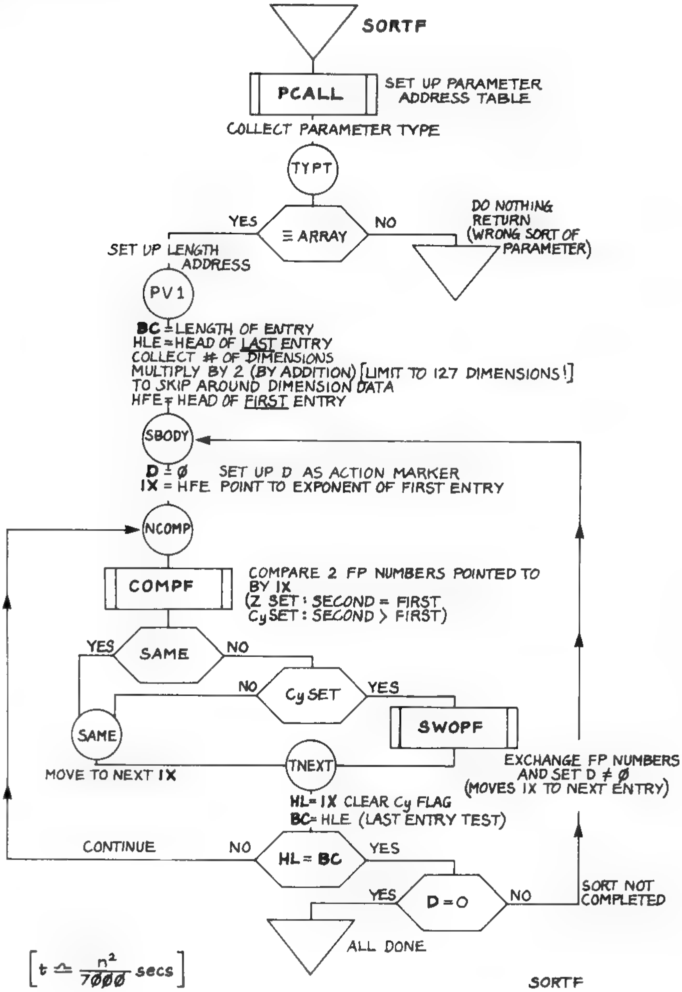
German proverb

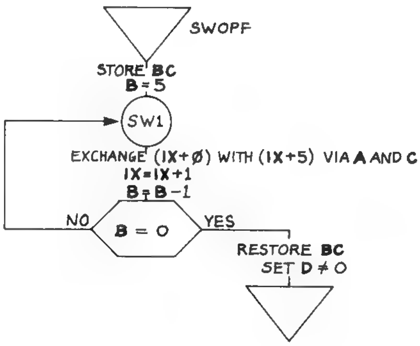
For those of the class who have struggled this far, let me present a useful, practical routine, the sorting of an array of Spectrum format floating point numbers. This is a simple bubble sort with no practical restrictions on the size of the array. The time for execution depends on the square of the number of entries and is roughly $n^2/7000$ seconds for n entries — some 125 times faster than the equivalent BASIC routine. For 1000 entries the sort would take about 145 seconds as against five hours.

Figure 10.1



Flowchart 10.1





The bubble sort is not the fastest but it is the simplest. Adjacent entries in the table are compared, and the larger, if it is not the earlier, swapped with the smaller: the whole table is repeatedly scanned until no inversions are made in a scan, at which point the table has been sorted and the routine exits.

The first pass from top to bottom, will always carry the lowest value to the bottom. If the next pass is made from bottom to top the highest value will be carried to the top. With such a process the length of the unsorted table is continuously reduced by one entry for each pass and the time required can be reduced by around 50%. I will leave you to do this — I have done the difficult bits, parameter passing and the comparison of the floating point (FP) numbers.

SORTF

The routine separates into two sections. First the call of PCALL to set up the parameter list and the extraction of the located parameter followed by its checking. If the parameter is not of type 4 (array of numbers), the routine exits having done nothing: then the variables HFE — head of first entry — and HLE — head of last entry — are set up. As we have no multiplication routine the skip over the array length data in the variable area is done by setting the one byte of 'number of dimensions' into A and then adding it to itself; the restriction being that A does not exceed 127! A multi-dimension array is treated as being a single dimension array — the highest value is placed in x(1,1,. . .)

The body of SORTF is straightforward. The D register is used to indicate that an inversion has taken place — set in SWOPF — and COMPF compares two adjacent numbers. COMPF exits with the Z flag set if the numbers are equal and the C flag is set if the second is larger than the first. The two FP numbers are adjacent to each other and the IX register points to the exponent byte with the lower address.

Listing 10.1

```
SORTF   CALL  PCALL
        LD    HL,(PARMO)
        LD    (TYPT+1),HL
TYPT    LD    A,(TYPT+1)
        AND   128+64+32
        CP    128
        RET   NZ
        INC   HL
```

```
                LD      (PV1+2),HL
        PV1     LD      BC,(PV1+2)
                ADD     HL,BC
                LD      BC,-3
                ADD     HL,BC
                LD      (HLE),HL
                LD      HL,(PARMO)
                LD      BC,3
                ADD     HL,BC
                LD      A,(HL)
                ADD     A
                LD      C,A
                LD      B,0
                ADD     HL,BC
                INC     HL
                LD      (HFE),HL
        SBODY   LD      D,0
                LD      IX,(HFE)
        NCOMP   CALL    COMPF
                JR      Z,SAME
                JR      NC,SAME
                CALL    SWOPF
                JR      TNEXT
        SAME    LD      BC,5
                ADD     IX,BC
        TNEXT   PUSH    IX
                POP     HL
                OR      A
                LD      BC,(HLE)
                SBC     HL,BC
                JR      NZ,NCOMP
                LD      A,D
                CP      0
                RET     Z
                JR      SBODY
        HFE     DEFW    0
        HLE     DEFW    0
        SWOPF   PUSH    BC
                LD      B,5
        SW1     LD      A,(IX+0)
                LD      C,(IX+5)
                LD      (IX+0),C
                LD      (IX+5),A
                INC     IX
```

```
             DJNZ  SW1
             POP   BC
             LD    D,1
             RET
```

COMPF

Figure 10.2 details the format of an FP number. The routine is quite complicated and might be much simplified. IX points to the first (exponent) byte of the first number whose mantissa is at IX + 1,2,3, & 4. The second number has its exponent at IX + 5 and its mantissa at IX + 6,7,8 & 9. The signs of the numbers are in IX + 1 and IX + 6. If they differ, the positive is greater than the negative. If they are of the same sign, their exponents are compared. In the Spectrum representation the exponents are all offset by 128 and the exponents may be compared and the carry flag tested. The significance depends however on the sign of the mantissa. With positive mantissas the larger exponent belongs to the larger FP number; with negative mantissas the larger exponent belongs to the smaller FP number. The B register is set non zero for negative mantissas.

Numbers with the same sign and equal exponents must be compared byte by byte until a discrepancy, if any, is detected. The signed bytes, when compared, must be tested by JP P,... or JP M,... operations as a carry from a borrow will only be set with a negative number. The remaining mantissa bytes can be tested on the carry flag as they are all unsigned. The significance of the decision at BTL or BTG is decided on the sign, as recorded in the B register, of the mantissa; failing to make this correction will result in the positive numbers being separated from the negative ones and both sets sorted in order of descending absolute (unsigned) size.

Listing 10.2

```
COMPF  LD    B,0
       BIT   7,(IX+1)
       JR    Z,CL1
CL2    BIT   7,(IX+6)
       JR    Z,V1LV2
CL3    LD    A,(IX+0)
       CP    (IX+5)
       JP    M,V1GV2
```

```
4055            JR    Z,CL4
4060            JR    V1LV2
4065 CL4        LD    B,255
4070            JR    XEQ
4075 CL1        BIT   7,(IX+6)
4080            JR    NZ,V1GV2
4085 CL5        LD    A,(IX+5)
4090            CP    (IX+0)
4095            JR    C,V1GV2
4100 CL6        JR    NZ,V1LV2
4105 XEQ        LD    A,(IX+1)
4110            CP    (IX+6)
4115            JR    Z,XEQM
4120            JP    P,BTG
4125            JR    BTL
4130 XEQM       LD    A,(IX+2)
4135            CP    (IX+7)
4140            JR    C,BTL
4145            JR    NZ,BTG
4150            LD    A,(IX+3)
4155            CP    (IX+8)
4160            JR    C,BTL
4165            JR    NZ,BTG
4170            LD    A,(IX+4)
4175            CP    (IX+9)
4180            JR    C,BTL
4185            JR    NZ,BTG
4190            RET
4195 V1GV2      LD    A,2
4200            CP    1
4205            RET
4210 V1LV2      LD    A,2
4215            CP    3
4220            RET
4225 BTL        BIT   1,B
4230            JR    NZ,V1GV2
4235            JR    V1LV2
4240 BTG        BIT   1,B
4245            JR    NZ,V1LV2
4250            JR    V1GV2
```

Flowchart 10.2

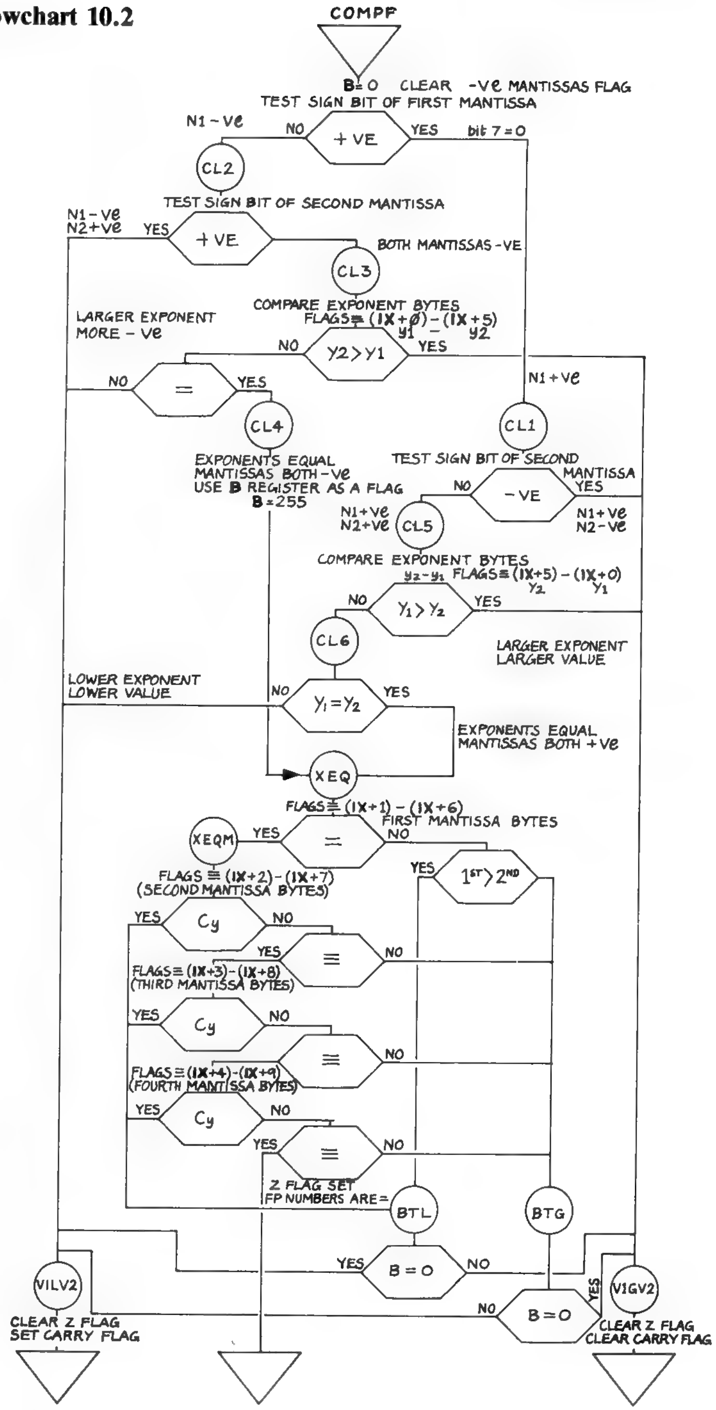
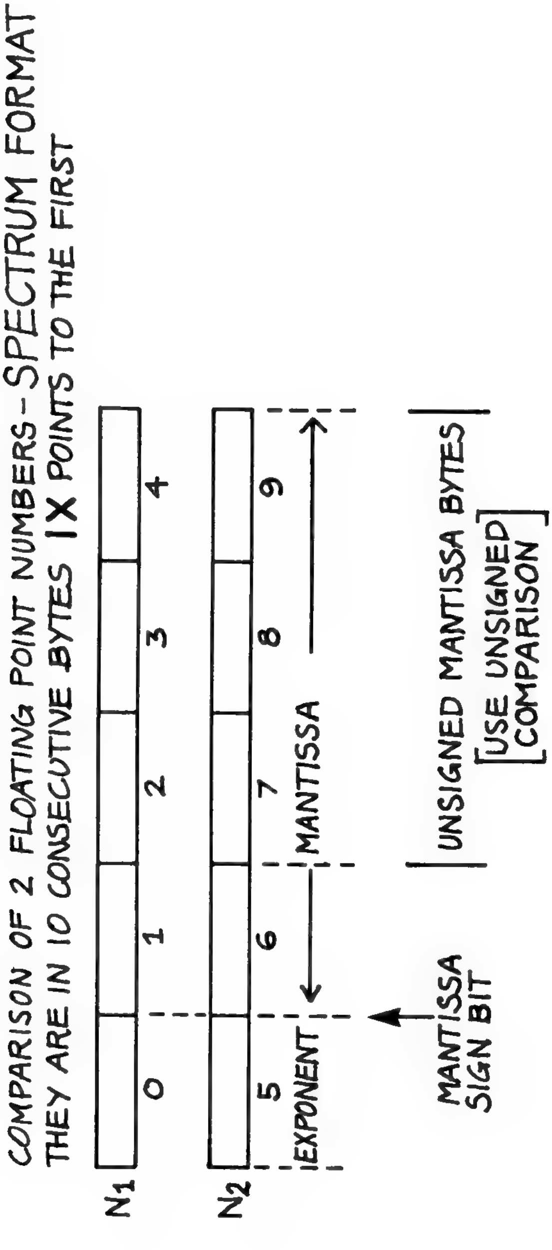


Figure 10.2


COMPARISON OF 2 FLOATING POINT NUMBERS - SPECTRUM FORMAT
THEY ARE IN 10 CONSECUTIVE BYTES IX POINTS TO THE FIRST
N1
N2
0
1
2
3
4
5
6
7
8
9
EXPONENT
MANTISSA
MANTISSA SIGN BIT
UNSIGNED MANTISSA BYTES
[USE UNSIGNED COMPARISON]

SORTF — a practical example

To print a list of competitors in order of descending points. (There are not more than 999 competitors and their points/times may be represented by a number less than six digits.

The data is in an array a( ), the points of competitor n are in a(n). If we sort a( ) we will indeed put the entries in numerical value order but we will lose the competitor identification.

If we code in BASIC:

```
FOR n = 1 TO...
LET a (n) = 1000* a (n) + n
NEXT n
```

then each entry will contain both pieces of information; as decimal digits the last three will identify the competitor and the others the points of that competitor. Note that because of the way the Spectrum handles floating point numbers the apparent value of a (n) should not be greater than about 1 000 000 000. We can now write:

```
LET l = USR SORTF
REM a( ):
```

and the array will be sorted with the higher marks first and the competitor number trailing along behind as the last three digits. The nth entry can then be printed by:

```
PRINT INT (a(n)/1000); INT(a(n) – 1000*INT(a(n)/1000))
```

There is just one other consideration. Some entries in a( ) may be stored in the integer form internally, which will upset SORTF. Before using it, each element must be in floating point form, and the best way to do this is to use something like

```
LET a(i) = a(i) + 65537 – 65537
```

For an M entry list the program looks like:

```
FOR n = 1 TO m
LET a (n) = 1000 * a (n) + n + 100 000
NEXT n
LET l = USR SORTF
REM a( ):
FOR n = 1 TO m
LET a (n) = a (n) – 100 000
PRINT INT(a(n)/1000); INT(a(n) – 1000*INT(a(n)/1000))
NEXT n
```

which will print the points followed by the competitor number. Where two competitors have the same number of points the output will be in descending competitor number order.

All you need to do is get the data into a( ) to start with and the routine will do the rest in the twinkling of an eye.

CHAPTER 11
Passing other Parameters

So far we have only a few routines which we call from the BASIC. It is quite easy to assemble each separately (with its own load address and its own copies of common subroutines), load them and remember to call them correctly as required. The drawbacks become evident when they have many subroutines in common and these are needlessly multiplied.

The solution I have adopted is shown in **Listing 11.1**. All the routines etc., are assembled together (or as many as are needed) and they are called through identical code sequences, which are thus all of the same length. First the SP save for the error escape back to line 2 (Chapter 9); then the storing of all the used registers, the specific subroutine call — DRAWL, MAP$ etc., and finally the jump to the common return label TRAP$ where the registers are restored and the RET to BASIC is made.

Listing 11.1

```
0005        ORG   60000
0010        LD    (ERROR),SP
0015        PUSH  AF
0020        PUSH  BC
0025        PUSH  DE
0030        PUSH  HL
0035        PUSH  IX
0040        CALL  DRAWL
0045        JP    TRAP$
0050        LD    (ERROR),SP
0055        PUSH  AF
0060        PUSH  BC
0065        PUSH  DE
0070        PUSH  HL
0075        PUSH  IX
0080        CALL  SATTR
0085        JP    TRAP$
0090        LD    (ERROR),SP
0095        PUSH  AF
```

```
0100        PUSH BC
0105        PUSH DE
0110        PUSH HL
0115        PUSH IX
0120        CALL BLOCK
0125        JP   TRAP$
0130        LD   (ERROR),SP
0135        PUSH AF
0140        PUSH BC
0145        PUSH DE
0150        PUSH HL
0155        PUSH IX
0160        CALL SORTF
0165        JP   TRAP$
0170        LD   (ERROR),SP
0175        PUSH AF
0180        PUSH BC
0185        PUSH DE
0190        PUSH HL
0195        PUSH IX
0200        CALL GCELL
0205        JP   TRAP$
0210        LD   (ERROR),SP
0215        PUSH AF
0220        PUSH BC
0225        PUSH DE
0230        PUSH HL
0235        PUSH IX
0240        CALL MAP$
0245        JP   TRAP$
0250        LD   (ERROR),SP
0255        PUSH AF
0260        PUSH BC
0265        PUSH DE
0270        PUSH HL
0275        PUSH IX
0280        CALL IVERT
0285        JP   TRAP$
0290        LD   (ERROR),SP
0295        PUSH AF
0300        PUSH BC
0305        PUSH DE
0310        PUSH HL
0315        PUSH IX
```

These common entries are all 16 bytes long and the routines can be called as an offset to the load address:

DRAWL at USR + 0
SATTR at USR + 16
BLOCK at USR + 32

and so on. (The head of) a BASIC program could then look like **Listing 11.2**. This has the advantage that, if the load address has to be changed, only line 10 needs attention and, after the initial setting up, the routines can be called by mnemonics instead of numerial values. (The routines SATTR and DRAWL are described in Chapters 13 and 14).

Listing 11.2

```
  1 LET error=0: GO TO 10
  2 PRINT "ERROR =";error: STOP
 10 LET base=60000
 11 LET drawl=base+0
 12 LET sattr=base+16
 13 LET block=base+32
 14 LET sortf=base+48
 15 LET gcell=base+64
 16 LET   map=base+80
 17 LET ivert=base+96
 18 LET movec=base+112
 19 LET svert=base+126
 20  LET drawa=base+144
 21 LET demo1=base+160
 22 LET demo2=base+176
 25 GO SUB 70: PAUSE 200: GO SUB 80: GO SUB
  200: GO SUB 300: GO SUB 400: GO SUB
9000: GO TO 21
 30 LET b=250
 41 DIM k$(b,2)
 42 FOR x=1 TO b
 43 LET k$(x,1)= CHR$ 255
 44 LET k$(x,2)= CHR$ 255
 45 NEXT x
 50 LET k=0
 51 LET o1=0
 53 LET l= USR movec
 54 REM read cursor postion
```

```
 56 PRINT  AT 0,0;"            ": PRINT  AT
 0,0;l: POKE 23560,255
 57 IF l=ol THEN  LET l=65535
 58  LET k=k+1
 59  LET k$(k,2)= CHR$  INT (l/256)
 60  LET k$(k,1)= CHR$  INT (l-256*( INT
(l/256)))
 61 IF k=1 THEN  GO TO 58
 62 LET m= USR drawa
 63 REM k$():
 64 LET ol=l
 65 PRINT  AT 0,6;k
 66 GO TO 53
 70 LET l= USR sattr
 71 REM :0,0,15,11,8,
 72 LET l= USR sattr
 73 REM :16,12,31,23,16,
 74 LET l= USR sattr
 75  REM :24,0,31,6,24,
 76 RETURN
 80 LET l= USR map
 81 RETURN
100 LOAD "" CODE : LOAD "" CODE : GO TO 1
200 DIM a(44)
201 FOR m=1 TO 44
202  LET a(m)= RND *10^( INT ((30* RND )-15))
203 NEXT m
204 GO SUB 220
205 LET l= USR sortf
206 REM a():
207 PAUSE 1
208 GO SUB 220
209 RETURN
220 CLS
221 FOR m=1 TO 44 STEP 2
222 PRINT a(m),a(m+1)
223 NEXT m
224 RETURN
300 PRINT  AT 3,5;"TILE COLOUR DEMO."
301 FOR k=0 TO 255
302 LET l=demo1
303 REM k:0,0,14,7,
304 NEXT k
310  RETURN
```

```
 400 DIM g(5)
 401 FOR g=1 TO 500
 402 LET g(5)= INT (255* RND )
 403 LET g(3)= INT (31* RND )
 404 LET g(4)= INT (23* RND )
 405 LET g(1)= INT (g(3)* RND )
 406 LET g(2)= INT (g(4)* RND )
 407 LET l= USR demo2
 408 NEXT g
 409 RETURN
9000 POKE 23675,0: POKE 23676,150
9001 FOR x=0 TO 224 STEP 2
9002 LET l= USR gcell
9003 POKE 38400,x
9004 NEXT x: RETURN
```

I now have some explaining to do — the REM statements in Listing 11.2. Back in Chapter 9 I showed how variable NAMES could be passed into machine code, but skipped over as too complicated for then, the passing of numeric values and strings. In Chapter 10 we used a passed array name to provide the required pointer(s) for SORTF. Now we will deal with the omissions of Chapter 10.

OPARS (Other Parameters)

These must be compatible with the name parameters collected by PCALL, that is, be present in the same REM line along with the name parameters. The easiest way is to rely on splitting the parameter list into two parts: first the names terminated by a colon and then, after the colon the values and strings. We allow the possibility that there are no name parameters but still insist on the colon as marking the start of the value / string part.

The specification for these parameters is:

Each entry, including the last, is terminated by a comma.

The REM statement is terminated by an ENTER token.

Values are unsigned, 16 bit integers. (Their values are to be found in the variables VPARO, VPARO + 2 etc.)

Strings are deliminated by double quotation marks (''), may be of any length and must be terminated by a comma after the closing quotes. A string must not contain double quotes. The address of the first character in each string is to be found in the variables SPARO, SPARO + 2, etc.

No data is passed concerning the relative positions of the values and strings in the parameter list; only their relative positions within each class are preserved.

To enable 0 to be passed as a value a subsidiary byte, SBITZ is used and has bits 7,6,... set according to whether VPARO, VPAR+2 etc. are valid.

OPARS will force error exits:

10 failed to find end of REM statement
11 non digit in number
12 too many parameters (more than 6)
13 false read of number
14 number greater than 65535

Listing 11.3

```
4255 OPARS  LD    HL,SPARO
4260        LD    (SPZ),HL
4265        LD    HL,VPARO
4270        LD    (VPZ),HL
4275        LD    HL,0
4280        LD    (SPARO),HL
4285        LD    HL,SPARO
4290        LD    DE,SPARO+1
4295        LD    BC,SBITZ-SPARO+1
4300        LDIR
4305        LD    HL,(NXTLN)
4310        INC   HL
4315        INC   HL
4320        LD    (VPL+2),HL
4325 VPL    LD    BC,(VPL+2)
4330        INC   HL
4335        INC   HL
4340 GNB1   CALL  GETBY
4345        CP    13
4350        RET   Z
4355        CP    ":"
4360        JR    NZ,GNB1
4365 GNB2   CALL  GETBY
4370        CP    13
4375        RET   Z
4380        CP    " "
```

```
               JR    Z,GNB2
               CP    ","
               JR    Z,EOPAR
               CP    """"
               JP    Z,STSTR
               CP    "0"
               JP    M,ERX11
               CP    ":"
               JP    P,ERX11
               JR    Z,EOPAR
               SUB   "0"
               PUSH  HL
               PUSH  BC
               OR    A
               LD    HL,(NUMB)
               ADD   HL,HL
               JR    C,ERX14
               ADD   HL,HL
               JR    C,ERX14
               LD    BC,(NUMB)
               ADD   HL,BC
               JR    C,ERX14
               ADD   HL,HL
               JR    C,ERX14
               LD    B,0
               LD    C,A
               ADD   HL,BC
               JR    C,ERX14
               LD    (NUMB),HL
               LD    A,1
               LD    (NNR),A
               POP   BC
               POP   HL
               JR    GNB2
     ERX14     LD    DE,14
               CALL  ERREX
     EOPAR     PUSH  HL
               PUSH  BC
               LD    A,(NNR)
               CP    0
               JP    Z,ERX13
               LD    A,(SBITZ)
               SRL   A
               SET   7,A
```

```
4605            LD    (SBITZ),A
4610            LD    HL,(NUMB)
4615            LD    BC,(VPZ)
4620            LD    (VPL2+1),BC
4625 VPL2       LD    (VPL2+1),HL
4630            INC   BC
4635            INC   BC
4640            LD    (VPZ),BC
4645            LD    HL,NUMB+2
4650            OR    A
4655            SBC   HL,BC
4660            JP    Z,ERX12
4665            LD    HL,0
4670            LD    (NUMB),HL
4675            LD    A,0
4680            LD    (NNR),A
4685            POP   BC
4690            POP   HL
4695            JP    GNB2
4700 STSTR      PUSH  HL
4705            PUSH  BC
4710            EX    DE,HL
4715            LD    BC,(SPZ)
4720            LD    HL,VPARO
4725            OR    A
4730            SBC   HL,BC
4735            JR    Z,ERX12
4740            EX    DE,HL
4745            LD    (VPL3+1),BC
4750 VPL3       LD    (VPL3+1),HL
4755            INC   BC
4760            INC   BC
4765            LD    (SPZ),BC
4770            POP   BC
4775            POP   HL
4780 RFORC      CALL  GETBY
4785            CP    """"
4790            JR    NZ,RFORC
4795 GNB3       CALL  GETBY
4800            CP    ","
4805            JP    Z,GNB2
4810            CP    13
4815            RET   Z
4820            JR    GNB3
```

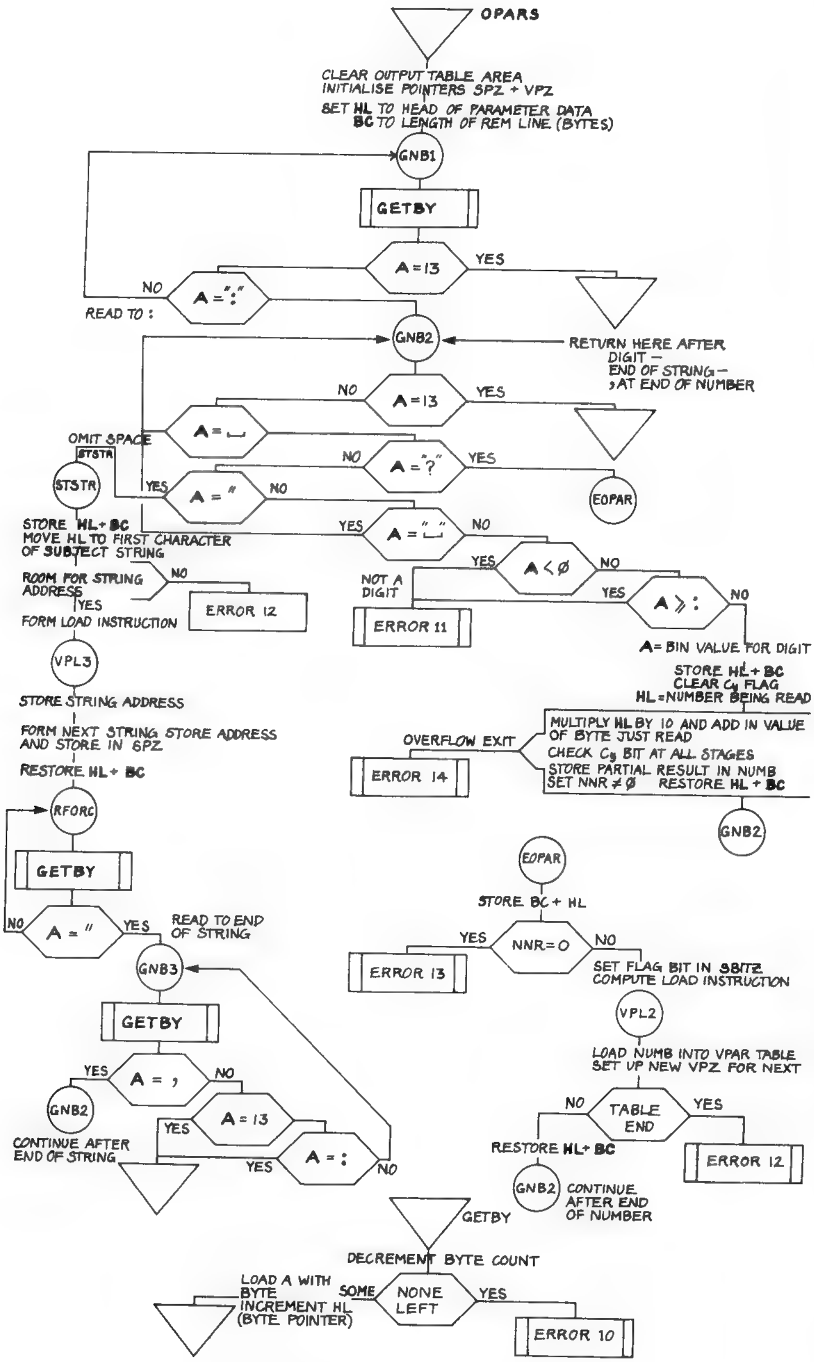
```
GETBY   DEC   BC
        BIT   7,B
        JR    NZ,ERX10
        LD    A,(HL)
        INC   HL
        RET
ERX10   LD    DE,10
ERXAA   CALL  ERREX
ERX11   LD    DE,11
        JR    ERXAA
ERX12   LD    DE,12
        JR    ERXAA
ERX13   LD    DE,13
        JR    ERXAA
VPZ     DEFW  0
SPZ     DEFW  0
SPARO   DEFW  0
        DEFW  0
        DEFW  0
        DEFW  0
        DEFW  0
        DEFW  0
VPARO   DEFW  0
        DEFW  0
        DEFW  0
        DEFW  0
        DEFW  0
        DEFW  0
NUMB    DEFW  0
NNR     DEFB  0
SBITZ   DEFB  0
        DEFW  0
```

Operation

As the routine may well be called many times all the workspace is first cleared and the result pointers, SPZ for strings and VPZ for values, are set to point to the heads of their respective lists, SPARO and VPARO.

At VPL BC is loaded with the number of characters in the parameter line following the call of USR... and HL is set up to point to the first byte. GETBY reads bytes in sequence using HL and decrementing BC (error 10 if BC goes negative) which are preserved for this use; the read character is in the A register.

Flowchart 11.1



At GNB1 characters are ready until either a 13 (ENTER token) or a : is met; the colon marks the end of the name parameter part which may be empty.

At GNB2, the characters after the colon are analysed; a 13 terminates the routine; spaces are ignored; a comma is recognised as an End Of PARameter (jump to EOPAR) and a double quote is recognised as the start of a string parameter to be dealt with at STart STRing (jump to STSTR). Anything left must be a (decimal) digit or an error.

Numbers

The ASCII codes for digits run sequentially from 48_{10} for 0 to 58_{10} for 9 and the colon has ASCII code 59_{10}. Subtracting the code for 0 leaves a valid binary representation of the digit just obtained.

The HL and BC registers are saved for their next use by GETBY and HL loaded with NUMB which holds the partial result of this value evaluation (or zero). HL is multiplied by 10 through shifting and addition and then A is added in to give the new partial result which is restored in NUMB. At each stage HL is tested for overflow and error 10 is generated if need be. NNR is set non-zero as an indicator that a number is being read and HL, BC are restored ready to read the next input byte.

End Of PARameter (EOPAR)

If NNR is not set, an error condition (double commas or missing value) raises error 13, otherwise a valid number has been read and a new bit is set in SBITZ. If the number were zero the VPAR entry would be zero. So a non-zero entry cannot be used as a test for the presence of an entry as it can be in SPAR for strings since 0 is head of memory in ROM. VPL2 is a computed load address for HL into the VPAR list and then VPZ is incremented by 2 to point to the next two byte entry. If it points to NUMB + 2 the table has overflowed and error 12 is generated. NUMB and NNR are cleared in readiness for the next value parameter.

N.B. The sequence of the labels VPZ to SBITZ should not be altered although the number of elements in the VPAR and SPAR lists may be changed.

String start (STSTR)

HL points to a byte just after the double quote which has been read by GETBY. HL and BC are stored, and HL — the address of the first character in the string — is stored in DE; OR A clears any carry flag and SPZ is tested against VPARO which marks the end of the SPAR list. Error 12 is again generated if there are too many string parameters. VPL3 is a computed load of the restored HL (from DE) into the string address table.

After the string address table has been loaded RFORC reads down the string for the terminating double quote and then to the concluding comma or terminating 13 token.

Synopsis

OPARS allows constants, integer values and strings to be passed into your machine code from the BASIC program. These parameters must follow a colon in the REM statement.

CHAPTER 12
BASIC Block Delete

If you wish to remove a section of lines from your BASIC program, because it has become obsolete for example, Then you normally have to type in each line number in turn, which can be very time consuming. Many other micros have a DELETE a,b or similar command which removes all lines from a to b. The following routine uses OPARS to delete any number of lines. It is best to refer to page 166 of the Spectrum manual while following this routine. It requires two value parameters, both line numbers, and deletes from the first up to, but not including, the second. The technique is one of individual line deletion followed by the adjustment of VARS. The BASIC system should be set up by CLEAR commands both before and, especially, after running the routine.

First some subroutines to collect individual lines for examination (see **Flowchart 12.1**). Note that they are essentially different ways of entering a common block of code.

SUPLN

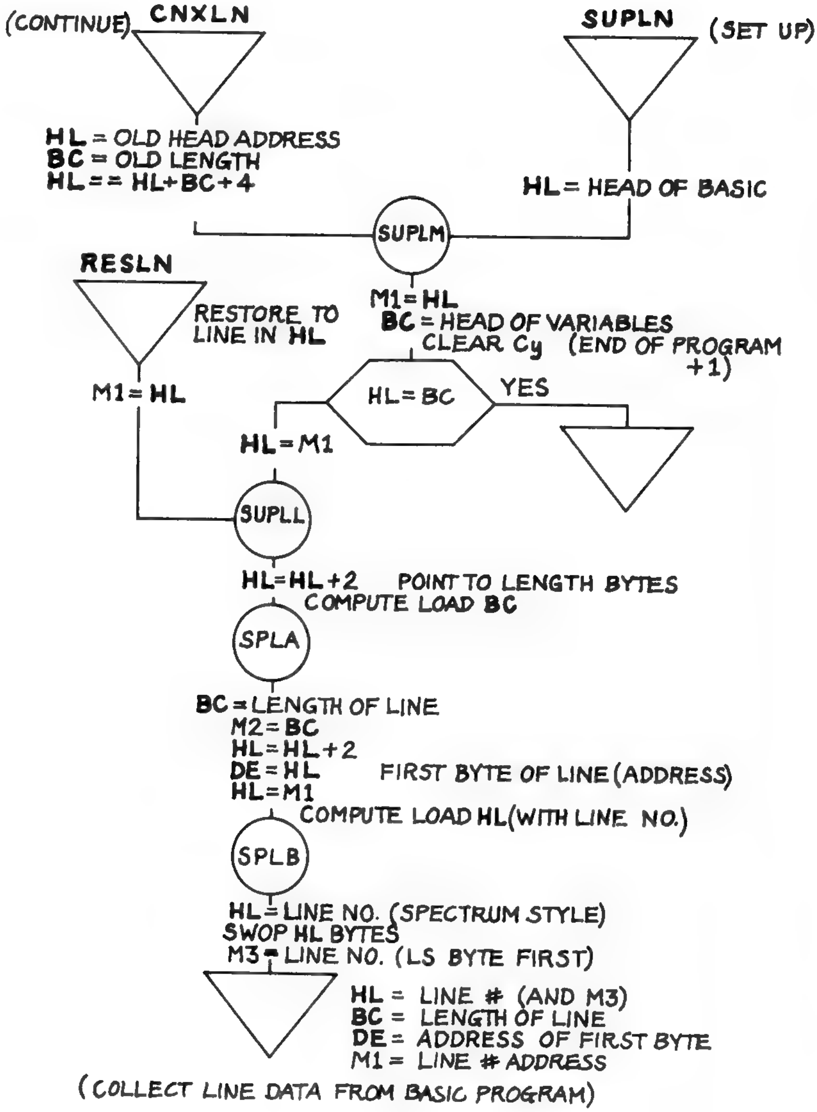
Sets UP LiNe pointers, used by the other routines, to point to the first line of the BASIC program; it and the others all destroy their input registers and exit as follows:

HL	contains the (new) line number
BC	the length, in bytes, of the line of data
DE	points to the first character of the line
Z flag	set if there is no more data

The variables M1, M2, M3, M4, and M5 are used as follows:

M1	address of first byte of line number
M2	length of this line in bytes (= BC)
M3	line number of this line (= HL)
M4–M5	temporary storage while a line is being deleted

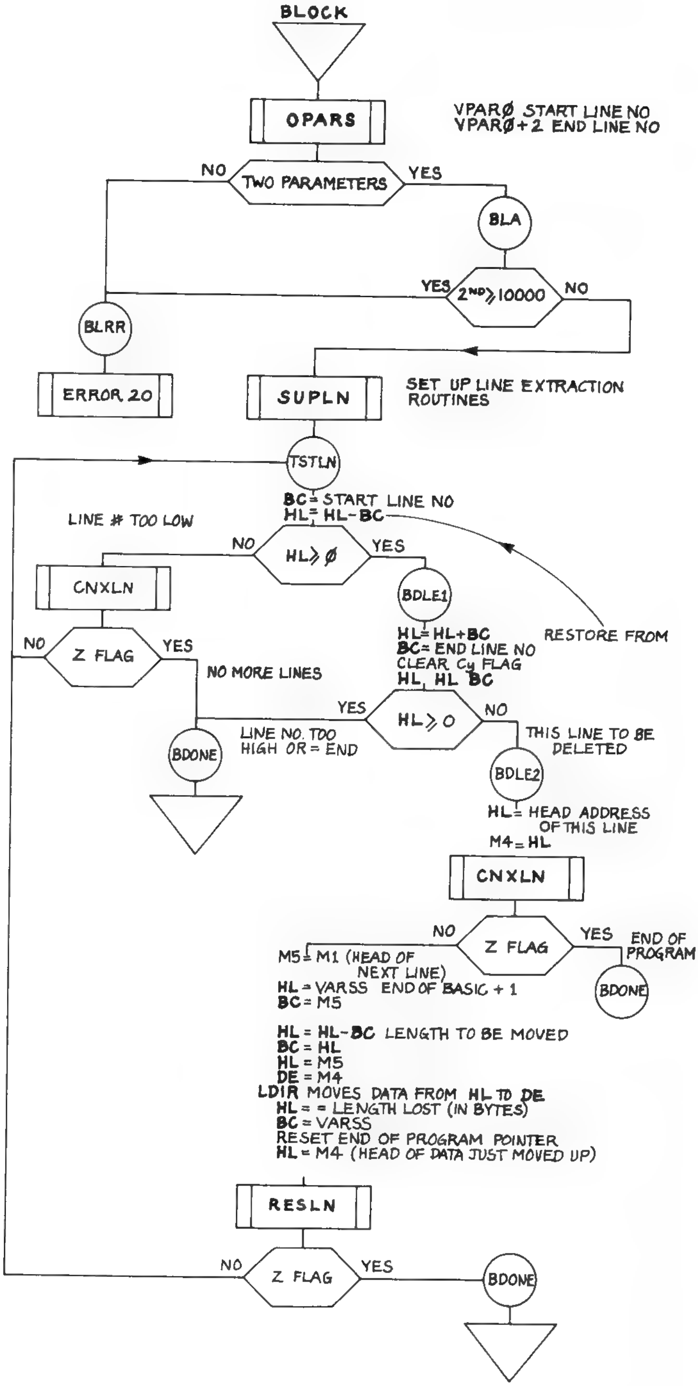
Flowchart 12.1



Listing 12.1

```
SUPLN   LD      HL,(PROG$)
SUPLM   LD      (M1),HL
        LD      BC,(VARSS)
        OR      A
        SBC     HL,BC
        RET     Z
        LD      HL,(M1)
SUPLL   INC     HL
        INC     HL
        LD      (SPLA+2),HL
SPLA    LD      BC,(SPLA+2)
        LD      (M2),BC
        INC     HL
        INC     HL
        EX      DE,HL
        LD      HL,(M1)
        LD      (SPLB+1),HL
SPLB    LD      HL,(SPLB+1)
        LD      A,H
        LD      H,L
        LD      L,A
        LD      (M3),HL
        RET
CNXLN   LD      BC,(M2)
        LD      HL,(M1)
        ADD     HL,BC
        INC     HL
        INC     HL
        INC     HL
        INC     HL
        JR      SUPLM
RESLN   LD      (M1),HL
        JR      SUPLL
M1      DEFW    0
M2      DEFW    0
M3      DEFW    0
M4      DEFW    0
M5      DEFW    0
PROG$   EQU     23635
```

Flowchart 12.2



CNXLN (Continue with NeXt LiNe)

This sets up the registers in a similar way to SUPLN, but for the next line in the program. It sets HL not to PROG, as SUPLN does, but to M1 + M2 + 4, which is the first byte of the next line.

RESLN (REStore LiNe)

After a line has been deleted the old line location(s) now contains the head of the next, non deleted line. RESLN resets the registers and storage locations for his new line.

Operation of SUPLN

On entry HL contains the location of a line, initially the first one. This address is stored in M1 and compared with the value of VARS. The RET Z will return if the HL has reached VARS ie there are no more lines.

At SUPLL, HL is incremented by two to point to the line length bytes and this value is stored in SPLA + 2, which is the second half of the next instruction. The computed instruction at SPLA loads BC with the length of the current line, and it is stored in M2. The computed instruction at SPLB then loads HL with the line number, which is reversed, so registers H and L are swopped, stored in M3 and a return made. In all normal circumstances the Z flag will be unset because no instruction apart from the SBC test after SUPLM will affect any flag. Take care that at the entry RESLN the Z flag is NOT set.

Operation of BLOCK (see Flowchart 12.2)

BLOCK expects two parameters and its call will look like:

```
LET L = USR...
REM : 174, 8234,
```

Should the second parameter be less than the first no action will take place. OPARS is called to read the two parameters which will be located in VPARO and VPARO + 2 as two 16 bit numbers.

SBITZ is checked to ensure that only two parameters are present (error 20 otherwise) and the value of the second parameter is checked to be a valid line number (less than 10000). SUPLN is now called to point to the first BASIC line and at TSTLN the line number is checked against the value of the first parameter; if the value is too small CNXLN is called to collect the next line and the process repeated whilst lines remain to be checked; if the line number is equal or greater than the first parameter a jump is made to BDLE1.

Listing 12.2

```
5180 BLOCK  CALL OPARS
5185        LD   A,(SBITZ)
5190        XOR  128+64
5195        JR   Z,BLA
5200 BLRR   LD   DE,20
5205        CALL ERREX
5210 BLA    LD   HL,(VPARO+2)
5215        LD   BC,10000
5220        OR   A
5225        SBC  HL,BC
5230        JP   P,BLRR
5235        CALL SUPLN
5240 TSTLN  LD   BC,(VPARO)
5245        OR   A
5250        SBC  HL,BC
5255        JP   P,BDLE1
5260        CALL CNXLN
5265        JR   Z,BDONE
5270        JR   TSTLN
5275 BDLE1  ADD  HL,BC
5280        LD   BC,(VPARO+2)
5285        OR   A
5290        SBC  HL,BC
5295        JP   P,BDONE
5300 BDLE2  LD   HL,(M1)
5305        LD   (M4),HL
5310        CALL CNXLN
5315        JR   Z,BDONE
5320        LD   HL,(M1)
5325        LD   (M5),HL
5330        LD   HL,(VARSS)
5335        LD   BC,(M5)
5340        OR   A
5345        SBC  HL,BC
5350        PUSH HL
5355        POP  BC
5360        LD   HL,(M5)
5365        LD   DE,(M4)
5370        LDIR
5375        OR   A
5380        LD   HL,(M5)
```

```
              LD    BC,(M4)
              SBC   HL,BC
              PUSH  HL
              POP   BC
              LD    HL,(VARSS)
              SBC   HL,BC
              LD     (VARSS),HL
              LD    HL,(M4)
              CALL  RESLN
              JR    NZ,TSTLN
      BDONE   RET
```

BDLE1

The line number is now checked against the second parameter, the upper line limit. If this is less than the limit the line is to be deleted at BDLE2, otherwise the routine exits at BDONE to the cruel, cold world of a diminished BASIC program.

BDLE2

This line is to be deleted. The head address is stored in M4 and CNXLN is called to determine the head address and presence of the next line. BLOCK specifically will *not* delete the last line of the BASIC program.

BC is loaded with the number of bytes to be retained (from the head of the line called up by CNXLN up to the address in VARS) and DE/HL are set so that the LDIR instruction will move everything up, so covering the unwanted line. VARS is then reduced by the total length of the removed line and RESLN called (with the Z flag not set).

The process is now repeated at TSTLN where the next line is tested and, if need be, deleted.

CHAPTER 13
Setting the Attributes Area

The attributes area controls the INK and PAPER colours and the BRIGHT and FLASHing status of each character square. They are arranged sequentially from location 22528, in the form of 24 rows of 32 columns. This routine allows you to set all or any of the attributes for a rectangular area by specifying the top left and bottom right hand tiles of the area involved, together with the required attribute(s) byte.

The call is

LET L = USR...
REM : X_t, Y_t, X_b, Y_b, A,

The Xs must be in the range 0–31 and the Ys in the range 0–23. The A value is the decimal number, collected from **Figure 13.1** which defines what is to happen at a tile position. Remember, you can disguise a messy screen redrawing by setting paper and ink colours the same to start with and then revealing all by setting them differently when done.

Two errors may be generated by the routine:

30 'top left' corner below or to the right of 'bottom right' corner
31 either specified tile is outside the attributes area

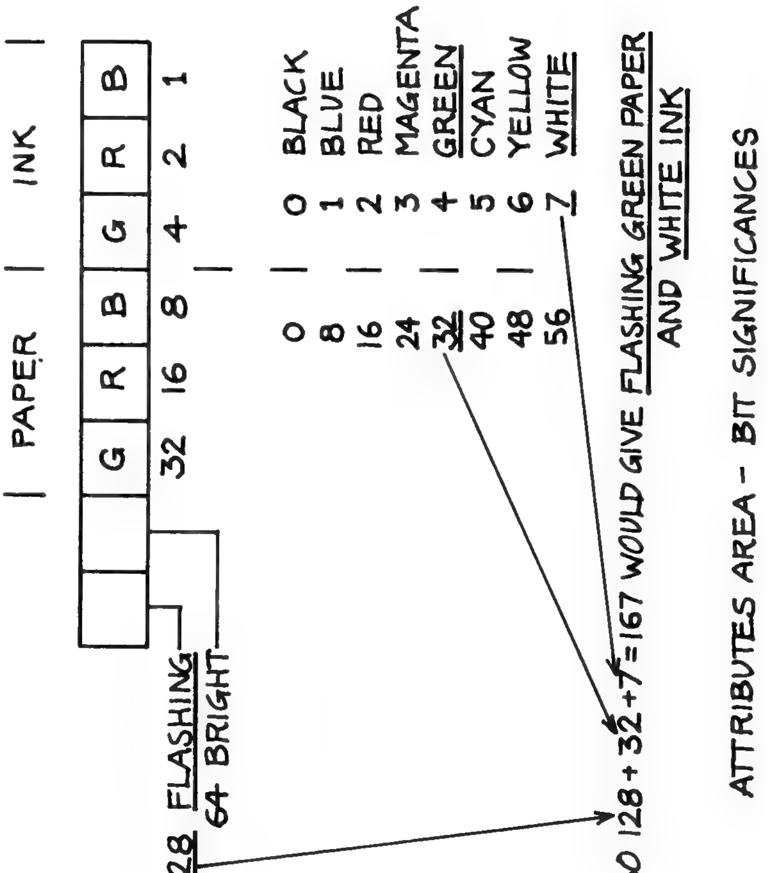
Operation (see Flowchart 13.1)

OPARS collects the value parameters which are assumed to be present, and STRTA is calculated to be the address of the first attributes byte to be loaded.

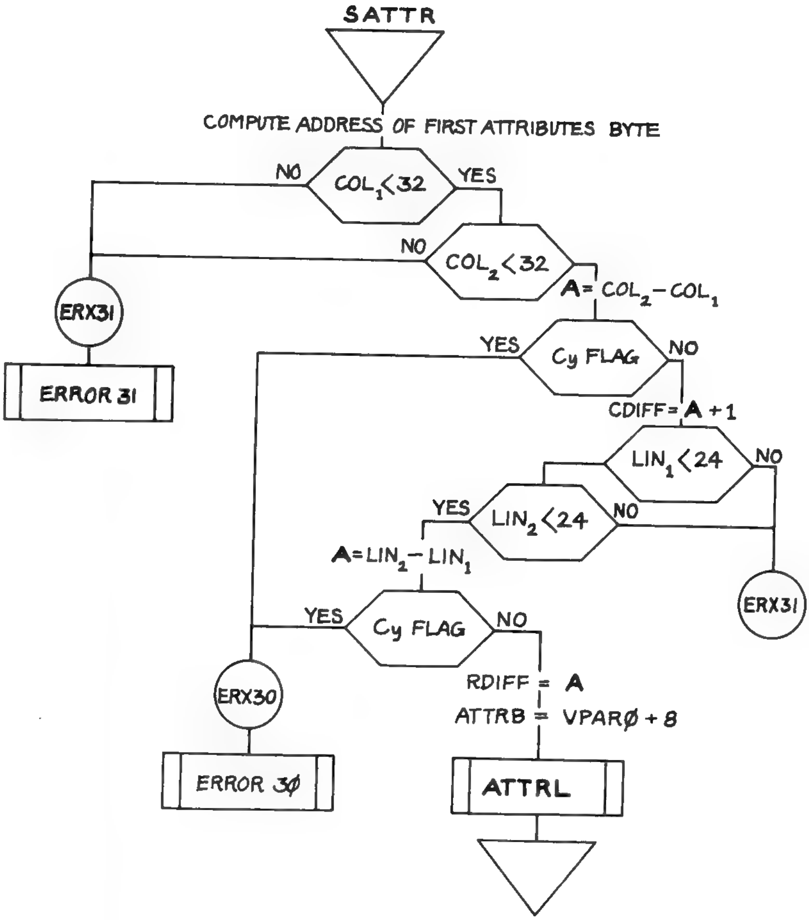
CDIFF holds the difference + 1 in the tile columns (X values) specified and RDIFF the row difference + 1. If either the row or column values are the same a single row or column will be handled. When several calls are made remember that where a bottom right corner of one call is the same as the top left corner of another there will be a one tile overlap with the later overwriting the earlier.

Once RDIFF and CDIFF have been set up the double loop in the routine ATTRL F 13 write RDIFF rows of CDIFF attributes; each row of attributes commences 32 bytes beyond the start of the previous row and there are none of the complications of pixel plotting to be dealt with.

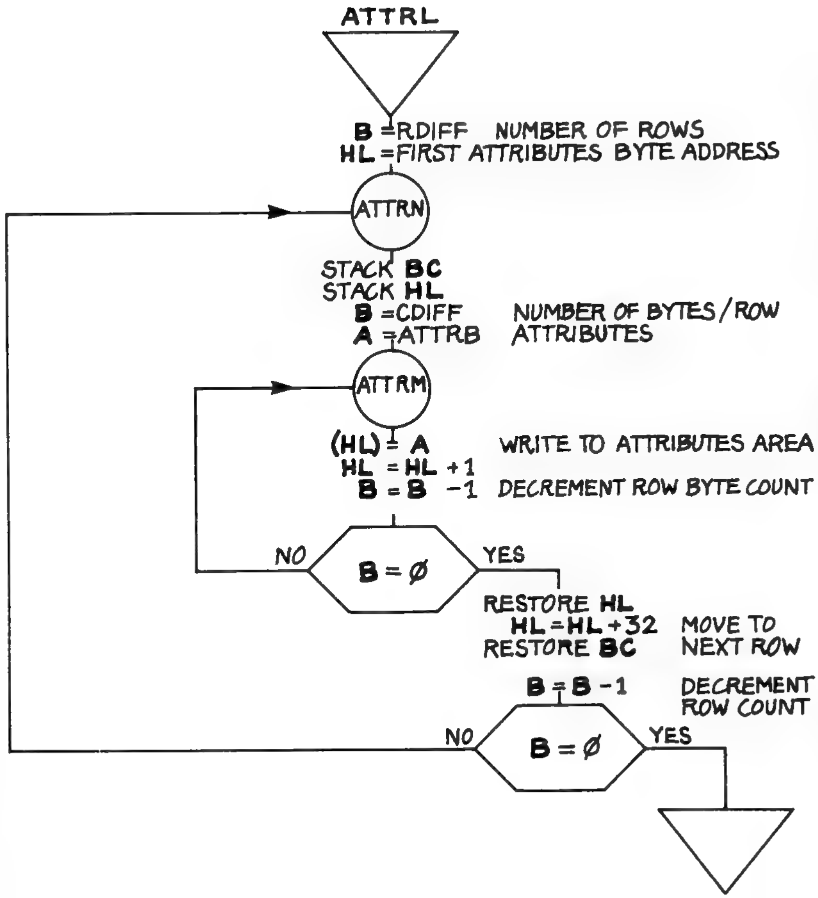
Figure 13.1


PAPER
INK
G R B G R B
32 16 8 4 2 1
128 FLASHING
64 BRIGHT
0 0 BLACK
8 1 BLUE
16 2 RED
24 3 MAGENTA
32 4 GREEN
40 5 CYAN
48 6 YELLOW
56 7 WHITE
SO 128 + 32 + 7 = 167 WOULD GIVE FLASHING GREEN PAPER AND WHITE INK
ATTRIBUTES AREA - BIT SIGNIFICANCES

Flowchart 13.1



Flowchart 13.2



Listing 13.1

```
5510 SATTR  CALL OPARS
5515        LD   HL,(VPARO+2)
5520        ADD  HL,HL
5525        ADD  HL,HL
5530        ADD  HL,HL
5535        ADD  HL,HL
5540        ADD  HL,HL
5545        LD   BC,(VPARO)
5550        ADD  HL,BC
5555        LD   BC,16384+6144
5560        ADD  HL,BC
5565        LD   (STRTA),HL
5570        LD   A,(VPARO)
5575        CP   32
5580        JP   P,ERX31
5585        LD   B,A
5590        LD   A,(VPARO+4)
5595        CP   32
5600        JP   P,ERX31
5605        SUB  B
5610        JR   C,ERX30
5615        INC  A
5620        LD   (CDIFF),A
5625        LD   A,(VPARO+2)
5630        CP   24
5635        JP   P,ERX31
5640        LD   B,A
5645        LD   A,(VPARO+6)
5650        CP   24
5655        JP   P,ERX31
5660        SUB  B
5665        JR   C,ERX30
5670        INC  A
5675        LD   (RDIFF),A
5680        LD   A,(VPARO+8)
5685        LD   (ATTRB),A
5690        CALL ATTRL
5695        RET
5700 ATTRB  DEFB 0
5705 CDIFF  DEFB 0
5710 RDIFF  DEFB 0
```

```
STRTA  DEFW O
ERX30  LD   DE,30
       CALL ERREX
ERX31  LD   DE,31
       CALL ERREX
```

Listing 13.2

```
ATTRL  LD   A,(RDIFF)
       LD   B,A
       LD   HL,(STRTA)
ATTRN  PUSH BC
       PUSH HL
       LD   A,(CDIFF)
       LD   B,A
       LD   A,(ATTRB)
ATTRM  LD   (HL),A
       INC  HL
       DJNZ ATTRM
       POP  HL
       LD   BC,32
       ADD  HL,BC
       POP  BC
       DJNZ ATTRN
       RET
```

CHAPTER 14
Hi Res Graphics

The Spectrum has a display resolution of 256 pixels horizontally by 192 vertically. In this chapter there are routines to draw lines and move a cursor across it, and an elementary drawing program is also presented.

The only way to draw a line between two points on the Spectrum display is to plot, point by point, all possible points on the line from X_1, Y_1 to X_2, Y_2 and preferably to do it quickly.

One way to do it which gives reasonable results is as follows: find increments DX and DY, not necessarily integer or positive, in X and Y which can be repeatedly added to X_1, Y_1 and which will cause X_1, Y_1 to move towards and reach X_2, Y_2. This is, in principle, what happens when you draw a line with a straight edge on graph paper.

Problems now arise. How are we to deal with the fractions when we have only dealt so far with integers? Fear not! The answer lies not with floating point numbers but with scaling.

Scaling is a very common technique in machine code programming for dealing with values outside the normal byte or word range of the machine. By way of example we will take the points in X and Y on the display screen to be given by 16 bit numbers; the MS byte will represent actual plottable points and the LS byte the fractional, non plottable, parts.

We take the arithmetic (ie signed) differences between the Xs and between the Ys and divide each by 256 (by changing the byte significance) to generate the differences DX and DY. This will always work as the largest difference between two Xs can only be 255, but we must remember to treat DX and DY as 16 bit values and propagate their sign bits through the MS byte. To reduce the plotting work to be done DX and DY are both shifted left until their most significant digit amounts to one quarter of a plotted point; more than this results in a ragged line, less takes longer, the choice is yours and you ought to experiment by modifying the routine SDIFF which sets up DX and DY before they are used.

Listing 14.1

```
5825 PLINE   CALL OPARS
5830         LD   A,(VPARO)
```

```
5835        LD    B,A
5840        LD    A,(VPARO+2)
5845        LD    C,A
5850        LD    A,(VPARO+4)
5855        LD    D,A
5860        LD    A,(VPARO+6)
5865        LD    E,A
5870 XLINE  LD    (YO+1),BC
5875        LD    (XO),BC
5880        LD    (Y1+1),DE
5885        LD    (X1),DE
5890        LD    A,0
5895        LD    (XO),A
5900        LD    (X1),A
5905        LD    (YO),A
5910        LD    (Y1),A
5915        LD    (DX),A
5920        LD    HL,(X1+1)
5925        LD    BC,(XO+1)
5930        OR    A
5935        SBC   HL,BC
5940        LD    (DX),HL
5945        LD    (OLDDX),HL
5950        LD    HL,(Y1+1)
5955        LD    BC,(YO+1)
5960        OR    A
5965        SBC   HL,BC
5970        LD    (DY),HL
5975        LD    (OLDDY),HL
5980        CALL  SDIFF
5985 NPOIN  LD    A,(YO+1)
5990        LD    B,A
5995        LD    D,A
6000        LD    A,(XO+1)
6005        LD    C,A
6010        LD    E,A
6015        CALL  PLOT
6020 INVPT  OR    (HL)
6025        LD    (HL),A
6030        CALL  LPOIN
6035        RET   Z
6040 GNXPT  LD    HL,(XO)
6045        LD    BC,(OLDDX)
6050        ADD   HL,BC
```

```
6055           LD    (X0),HL
6060           LD    HL,(Y0)
6065           LD    BC,(OLDDY)
6070           ADD   HL,BC
6075           LD    (Y0),HL
6080           LD    A,(Y0+1)
6085           CP    D
6090           JR    NZ,NPOIN
6095           LD    A,(X0+1)
6100           CP    E
6105           JR    Z,GNXPT
6110           JR    NPOIN
6115 LPOIN     LD    A,(X0+1)
6120           LD    B,A
6125           LD    A,(X1+1)
6130           CP    B
6135           RET   NZ
6140           LD    A,(Y0+1)
6145           LD    B,A
6150           LD    A,(Y1+1)
6155           CP    B
6160           RET
6165 Y0        DEFW  0
6170 X0        DEFW  0
6175 Y1        DEFW  0
6180 X1        DEFW  0
6185 DX        DEFW  0
6190 DY        DEFW  0
6195 OLDDX     DEFW  0
6200 OLDDY     DEFW  0
6205 SDIFF     LD    A,(DX)
6210           LD    B,A
6215           LD    A,(DY)
6220           CP    B
6225           RET   Z
6230 SDIFH     LD    HL,(DX)
6235 SDIFG     LD    A,H
6240           CP    0
6245           JR    Z,SDIFA
6250           CP    255
6255           RET   NZ
6260 SDIFA     LD    HL,(DY)
6265           LD    A,H
6270           CP    0
```

```
6275          JR    Z,SDIFB
6280          CP    255
6285          RET   NZ
6290 SDIFB    ADD   HL,HL
6295          LD    (DY),HL
6300          SRA   H
6305          RR    L
6310          LD    (OLDDY),HL
6315          LD    HL,(DX)
6320          ADD   HL,HL
6325          LD    (DX),HL
6330          SRA   H
6335          RR    L
6340          LD    (OLDDX),HL
6345          JR    SDIFH
```

Operation of PLINE

As written, PLINE expects four value parameters in the REM statement, specifying the X_1, Y_1 and X_2, Y_2 points between which the line is to be drawn/plotted. These values are collected by OPARS and loaded without checking for validity into BC and DE.

XLINE

XLINE is another entry into the routine used by DRAWL, see below, which draws a series of lines. B, C, D, E are loaded into the MS bytes of X_0, Y_0, X_1 and Y_1 and the LS bytes are cleared. Observe carefully how the storage is arranged and *do not disturb* otherwise more instructions will be needed.

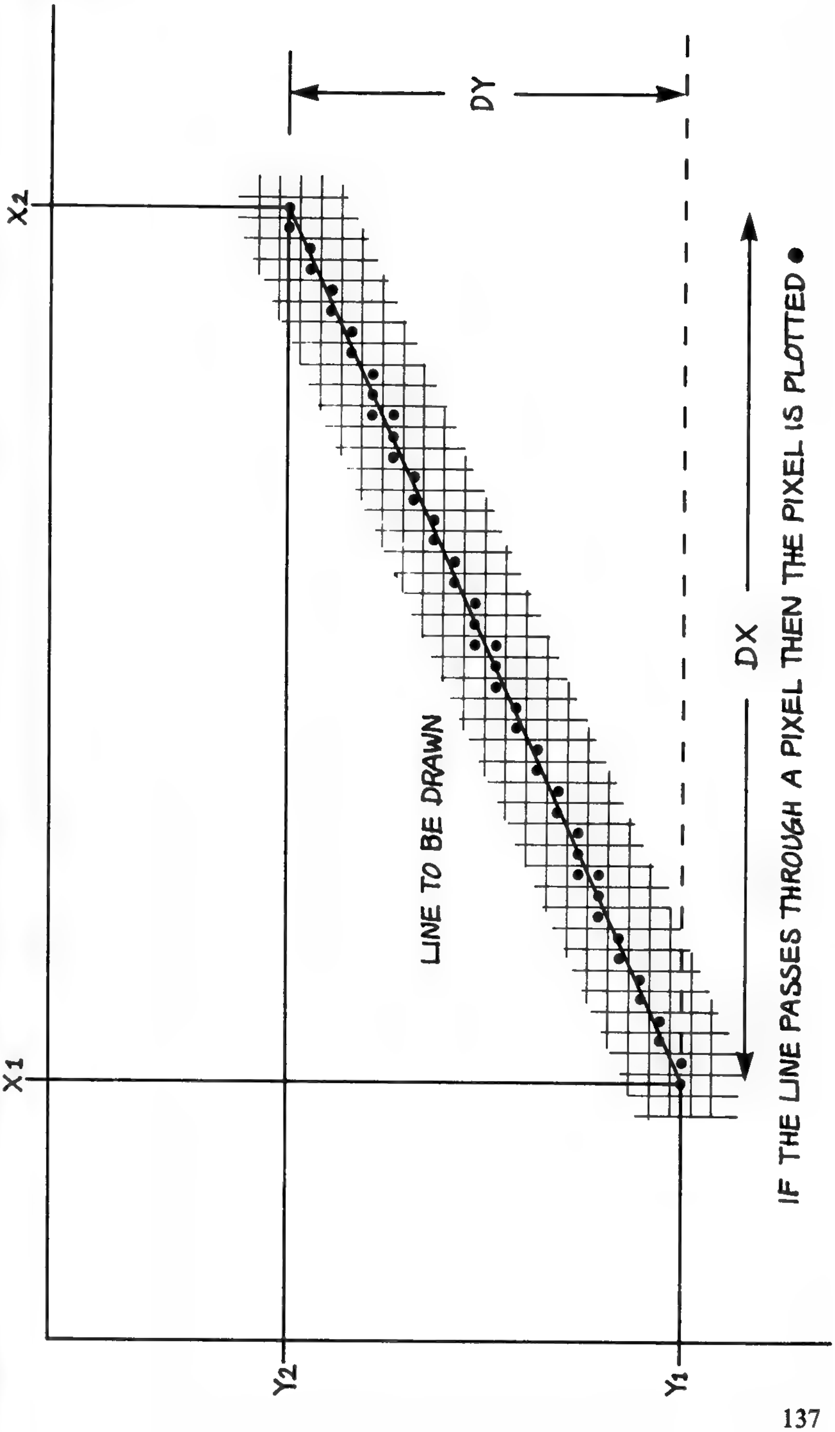
X_1 and X_0 are loaded into the low order bytes of HL and BC, the high order bytes are zero and DX is a 16 bit signed value formed from X_1-X_0; the high order byte is either zero or all 1s.

Similarly DY is set up from Y_1-Y_0. The subroutine SDIFF makes the values of DX and DY as large as possible but not more than one quarter of a pixel step and puts the vaues in OLDDX and OLDDY.

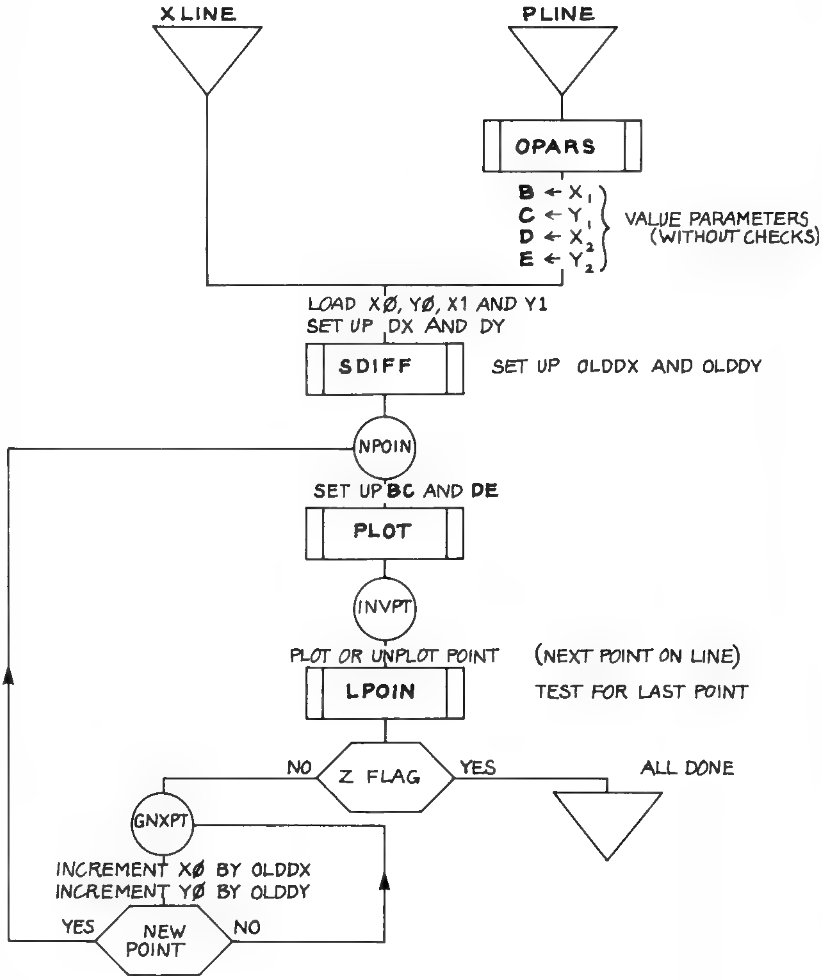
NPOIN

Here the next point is plotted. BC (and DE) are loaded with the coordinates Y in B, X in C and PLOT called. INVPT, which may be an OR or XOR instruction, modifies the contents of the display buffer. X_0 and Y_0, as 16 bit numbers, are incremented by the fractional values on OLDDX and OLDDY until either the new X_0 or Y_0 differs from the old as stored in DE.

Figure 14.1


DY
X2
X1
Y2
Y1
LINE TO BE DRAWN
DX
IF THE LINE PASSES THROUGH A PIXEL THEN THE PIXEL IS PLOTTED ●

Flowchart 14.1



This newly computed point is now plotted and so on until the plotted point coincides with X_1, Y_1 at which point the subroutine NPG returns with the Z flag set.

SDIFF

This happened fairly piecemeal and can be much improved.

DX and DY can be shifted left as long as their MS bytes remain either all 0s or all 1s and then right two places.

Now we can draw a line between two points. You probably won't use it at all because the next stage is more interesting.

Listing 14.2

```
6350 DRAWL  CALL OPARS
6355        LD   HL,(SPARO)
6360        CALL GVAL8
6365        RET  C
6370        LD   B,A
6375        CALL GVAL8
6380        RET  C
6385        LD   C,A
6390 DRNXP  CALL GVAL8
6395        RET  C
6400        LD   D,A
6405        CALL GVAL8
6410        RET  C
6415        LD   E,A
6420        PUSH HL
6425        CALL XLINE
6430        POP  HL
6435        PUSH DE
6440        POP  BC
6445        JR   DRNXP
6450 GVAL8  PUSH BC
6455        PUSH DE
6460 NBY    LD   A,(HL)
6465        INC  HL
6470        CP   """"
6475        JR   Z,JRCX
6480        CP   ","
6485        JR   Z,JRVX
6490        SUB  "0"
6495        LD   B,A
```

```
        LD    A,(BYTEV)
        SLA   A
        SLA   A
        LD    C,A
        LD    A,(BYTEV)
        ADD   C
        SLA   A
        ADD   B
        LD    (BYTEV),A
        JR    NBY
JRVX    LD    A,(BYTEV)
        LD    B,A
        LD    A,0
        LD    (BYTEV),A
        LD    A,B
        OR    A
        POP   DE
        POP   BC
        RET
JRCX    SCF
        POP   DE
        POP   BC
        RET
BYTEV   DEFB  0
```

DRAWL: Draw a list of lines

DRAWL has but one parameter, a string whose contents is a list of digits and commas which are interpreted as being X, Y pairs and the routine draws from pair 1 to pair 2 to pair 3 and so on to the end of the list. Note again that there are no validity checks on the sizes of the values except that GVAL8 only passes the LS 8 bytes of whatever value it finds; these checks can be inserted if you need them.

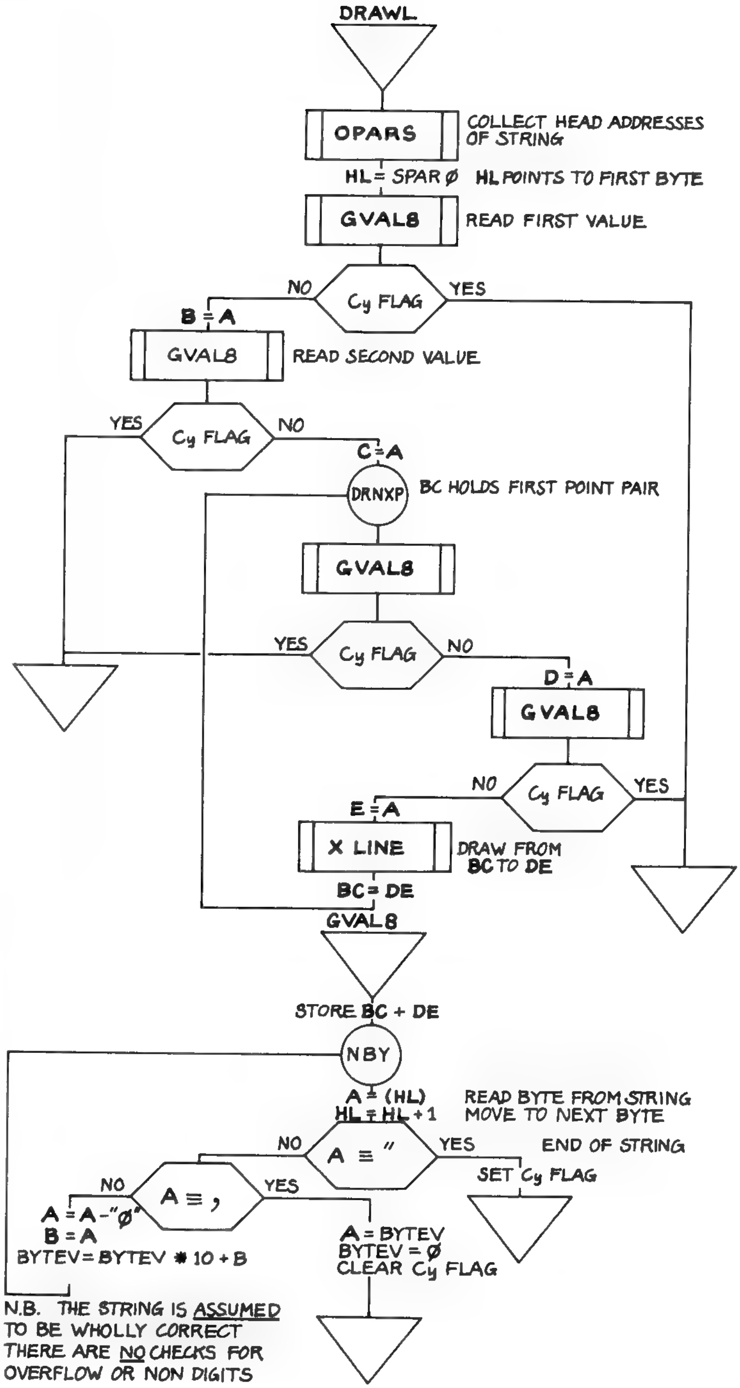
OPARS collects one string parameter and then GVAL8 recovers byte values from the string in a very primitive manner to load C, B, E, D for the call of XLINE to draw a line from BC to DE.

DE is transferred into DE and DE loaded with the next point position and the line BC to DE drawn; the process continues till GVAL8 exits with the carry flag set as a result of the exhaustion of the data string.

GVAL8

This is entered with HL pointing into the parameter string; A is loaded with the next character which is assumed to be:

Flowchart 14.2



a " marking the end of the string,
or a, marking the end of a value
or a digit. Non digits are not rejected but macerated.

BYTEV: BYTe EValuated

This is formed by shifting and adding to multiply by 10 and then adding in the binary value of the character, assumed to be a digit. There are no checks and the process continues till a comma is read.

Now we can draw lines what about undrawing them?

This is not too difficult. Change the OR (HL) at INVPT (invert plot) to XOR (HL) and all will be well so long as we retrace our steps precisely. Since there are, or may be, many points where this change is to be made the subroutine IVERT contains what amounts to a list of bytes which are to be changed. Repeated calls of IVERT change backwards and forwards; for those of us who get lost there is SVERT which sets all such options to OR for draw.

Listing 14.3

```
IVERT   LD    A,(INVPT)
        LD    B,A
        LD    A,(CHNGE)
        LD    (INVPT),A
        LD    (INVRX),A
        LD    A,B
        LD    (CHNGE),A
        RET
CHNGE   XOR   (HL)
SVERT   LD    A,(INVPT)
        LD    B,A
        LD    A,(XOROP)
        CP    B
        RET   NZ
        CALL  IVERT
        RET
XOROP   XOR   (HL)
```

MOVEC

This move cursor routine operates by plotting and unplotting a diamond of points. For faster movement you must either increment the cursor position by more than one pixel step or flit from tile to tile.

The basis is an IFKEY call which operates as follows:

5,6,7 & 8 keys move the cursor in the obvious directions

- s — sets slow movement
- f — sets fast movement
- x — sets single step movement
- p — causes the routine to exit, and yield up the cursor position. These keys are all in lower case.

A call LET L = USR. . . assigns to L, when the p key is operated, the current cursor position which may be unravelled by the BASIC program; prolonged depression of the key causes repeated outputs of the same position. On the first call the cursor is positioned near centre screen but repeated calls pick up the cursor from its last known position.

Listing 14.4

```
6705 MOVEC  CALL  SVERT
6710        CALL  CURSR
6715        CALL  IVERT
6720 MFAST  LD    A,1
6725        LD    (23561),A
6730        LD    (23562),A
6735 CIFKE  CALL  IFKEY
6740        DEFB  "8"
6745        JP    MRGHT
6750        DEFB  "5"
6755        JP    MLEFT
6760        DEFB  "7"
6765        JP    MUPUP
6770        DEFB  "6"
6775        JP    MDOWN
6780        DEFB  "p"
6785        JP    MEXIT
6790        DEFB  "f"
6795        JP    MFAST
6800        DEFB  "s"
6805        JP    MSLOW
6810        DEFB  "x"
6815        JP    MSTEP
6820        NOP
6825 MSTEP  LD    A,255
6830        LD    (23561),A
```

```
6835 MFASU  LD    (23562),A
6840        JR    CIFKE
6845 MSLOW  LD    A,10
6850        JR    MFASU
6855 MRGHT  LD    A,(CURSX)
6860 MRL1   INC   A
6865        CP    253
6870        JR    Z,MRLA
6875        LD    (CURSX),A
6880        JR    CIFKP
6885 MLEFT  LD    A,(CURSX)
6890 MRLA   DEC   A
6895        CP    2
6900        JR    Z,MRL1
6905        LD    (CURSX),A
6910        JR    CIFKP
6915 MUPUP  LD    A,(CURSY)
6920 MUPL1  DEC   A
6925        CP    2
6930        JR    Z,MDWNA
6935        LD    (CURSY),A
6940        JR    CIFKP
6945 MDOWN  LD    A,(CURSY)
6950 MDWNA  INC   A
6955        CP    188
6960        JR    Z,MUPL1
6965        LD    (CURSY),A
6970        JR    CIFKP
6975 CURSX  DEFB 125
6980 CURSY  DEFB 88
6985 CIFKP  CALL CROSS
6990        CALL IVERT
6995        CALL CURSR
7000        CALL IVERT
7005        JR    CIFKE
7010 PLOA   DEFB 254
7015        DEFB 254
7020 PLOB   DEFB 254
7025        DEFB +2
7030 PLOC   DEFB +2
7035        DEFB +2
7040 PLOD   DEFB +2
7045        DEFB 254
7050 PLOAT  DEFW 0
```

```
7055 PLOBT   DEFW O
7060 PLOCT   DEFW O
7065 PLODT   DEFW O
7070 CURSR   LD   BC,(CURSX)
7075         LD   HL,(PLOA)
7080         ADD  HL,BC
7085         LD   (PLOAT),HL
7090         LD   HL,(PLOB)
7095         ADD  HL,BC
7100         LD   (PLOBT),HL
7105         LD   HL,(PLOC)
7110         ADD  HL,BC
7115         LD   (PLOCT),HL
7120         LD   HL,(PLOD)
7125         ADD  HL,BC
7130         LD   (PLODT),HL
7135         CALL CROSS
7140         RET
7145 CROSS   LD   BC,(PLODT)
7150         CALL XPLOT
7155         LD   BC,(PLOBT)
7160         CALL XPLOT
7165         LD   BC,(PLOAT)
7170         CALL XPLOT
7175         LD   BC,(PLOCT)
7180         CALL XPLOT
7185         RET
7190 XPLOT   CALL PLOT
7195 INVRX   OR   (HL)
7200         LD   (HL),A
7205         RET
7210 MEXIT   CALL SVERT
7215         POP  IX
7220         POP  HL
7225         POP  DE
7230         POP  BC
7235         LD   BC,(CURSX)
7240         LD   A,5
7245         LD   (23562),A
7250         LD   A,35
7255         LD   (23561),A
7260         JP   TRAT$
```

Operation of MOVEC

The routine is so simple that by now you should not need a flow diagram but be able to work directly from the listing.

SVERT sets the plotting routine to a known state and then the Spectrum variables REPDEL and REPPER are set to their minimum values to give the fastest possible movement and the initial cursor position is plotted by a call on CURSR. IVERT is then called so that the next call will unplot the cursor diamond before plotting the second cursor position, this gives free non streaking movement.

The routine IFKEY now waits until a lower case menu key is read; the cursor keys 5, 6, 7, 8 cause jumps to MLEFT, MRGHT, MUPUP and MDOWN where the cursor position bytes, CURSX and CURSY are modified appropriately and then prevented from running off the screen; the old position is unplotted and the new plotted before the return to CIFKE for the next key operation.

The x, s, and f keys arrange for REPPER to be loaded with the appropriate values. Note here that one pixel vertically covers three television scan lines.

Other details

PLOA, PLOB, PLOC, and PLOD define the four diamond points with respect to the cursor position so that the actual cursor plot points may be obtained by the addition of CURSX, considered with CURSY, as a two byte value to these four points. These additions give the points PLOAT, PLOBT, PLOCT, and PLODT which are then plotted/unplotted by CROSS and XPLOT (according to the state of INVRX which is set by IVERT or SVERT).

When p is pressed, the routine exits through MEXIT which restores all the registers, except BC which it sets to the CURSOR position. As is usual with my routines the positions of byte/word declarations is important.

DRAWA: Draw Array

With this subroutine and MOVEC you can build a simple drawing program as sketched out in **Listing 14.5b**.

DRAWL looks for its data as point values in a REM parameter list. DRAWA is a variant on the same theme but this time the data is to be found in a two dimensional byte array which must be defined as:

DIM ?$(. . . , 2)

where ? is any suitable array reference and . . . is as large as need be. The character pair ?$(p,1) and ?$(p,2) contain the x and y plot values for the point p as one byte values. If the y value is off screen the point is omitted;

this enables a line sequence to be broken as required. The insertion of the off screen marker is a matter of convenience.

Listing 14.5a

```
7265 DRAWA  CALL PCALL
7270        LD   HL,(PARMO)
7275        LD   (DPL+1),HL
7280 DPL    LD   A,(DPL+1)
7285        AND  128+64+32
7290        CP   128+64
7295        JR   Z,DL1
7300 ERX40  LD   DE,40
7305        CALL ERREX
7310 DL1    LD   HL,(DPL+1)
7315        INC  HL
7320        LD   (DPLB+2),HL
7325 DPLB   LD   BC,(DPLB+2)
7330        INC  HL
7335        INC  HL
7340        LD   A,(HL)
7345        CP   2
7350        JR   Z,DL2
7355 ERX41  LD   DE,41
7360        CALL ERREX
7365 DL2    INC  HL
7370        INC  HL
7375        INC  HL
7380        LD   A,(HL)
7385        CP   2
7390        JR   NZ,ERX42
7395        INC  HL
7400        LD   A,(HL)
7405        CP   0
7410        JR   NZ,ERX42
7415        INC  HL
7420        PUSH HL
7425        LD   HL,-6
7430        ADD  HL,BC
7435        PUSH HL
7440        POP  BC
7445        POP  HL
7450 NXPPR  LD   (DPLC+2),HL
```

```
             INC   HL
             INC   HL
             LD    (DPLD+2),HL
             DEC   BC
             DEC   BC
             BIT   7,B
             RET   NZ
             PUSH  BC
             PUSH  HL
      DPLC   LD    BC,(DPLC+2)
             LD    A,B
             LD    B,C
             LD    C,A
      DPLD   LD    DE,(DPLD+2)
             LD    A,D
             LD    D,E
             LD    E,A
             LD    A,E
             AND   128+64
             CP    128+64
             JR    Z,EXT
             LD    A,C
             AND   128+64
             CP    128+64
             JR    Z,EXT
             CALL  XLINE
      EXT    POP   HL
             POP   BC
             JR    NXPPR
      ERX42  LD    DE,42
             CALL  ERREX
```

Listing 14.5b

```
30 LET b=250
41 DIM k$(b,2)
42 FOR x=1 TO b
43 LET k$(x,1)= CHR$ 255
44 LET k$(x,2)= CHR$ 255
45 NEXT x
50 LET k=0
51 LET ol=0
53 LET l= USR movec
```

```
54 REM read cursor postion
56 PRINT  AT 0,0;"                ": PRINT  AT 0,
0;l: POKE 23560,255
57 IF l=ol THEN  LET l=65535
58  LET k=k+1
59  LET k$(k,2)= CHR$  INT (l/256)
60  LET k$(k,1)= CHR$  INT (l-256*( INT (l/
256)))
61 IF k=1 THEN  GO TO 58
62 LET m= USR drawa
63 REM k$():
64 LET ol=l
65 PRINT  AT 0,6;k
66 GO TO 53
```

Operation of DRAWA

PCALL collects the parameter REM statement and the first parameter only is used. It is checked to be a character array exactly as specified; error 40 if not a character array, error 41 if not two dimensional and error 42 if the second dimension is not two.

At NXPPR the next (or first) point pair is obtained.

HL points to the first byte pair, DPLC is a computed load instruction, HL is moved on two bytes and DPLD is computed to load the next pair into DE. This will be the first byte pair next time round.

The byte pairs BC and DE must be swapped around for the call of XLINE. The swapping could be omitted but then the point pairs in the array parameter would need to be reversed and this is not the normal convention.

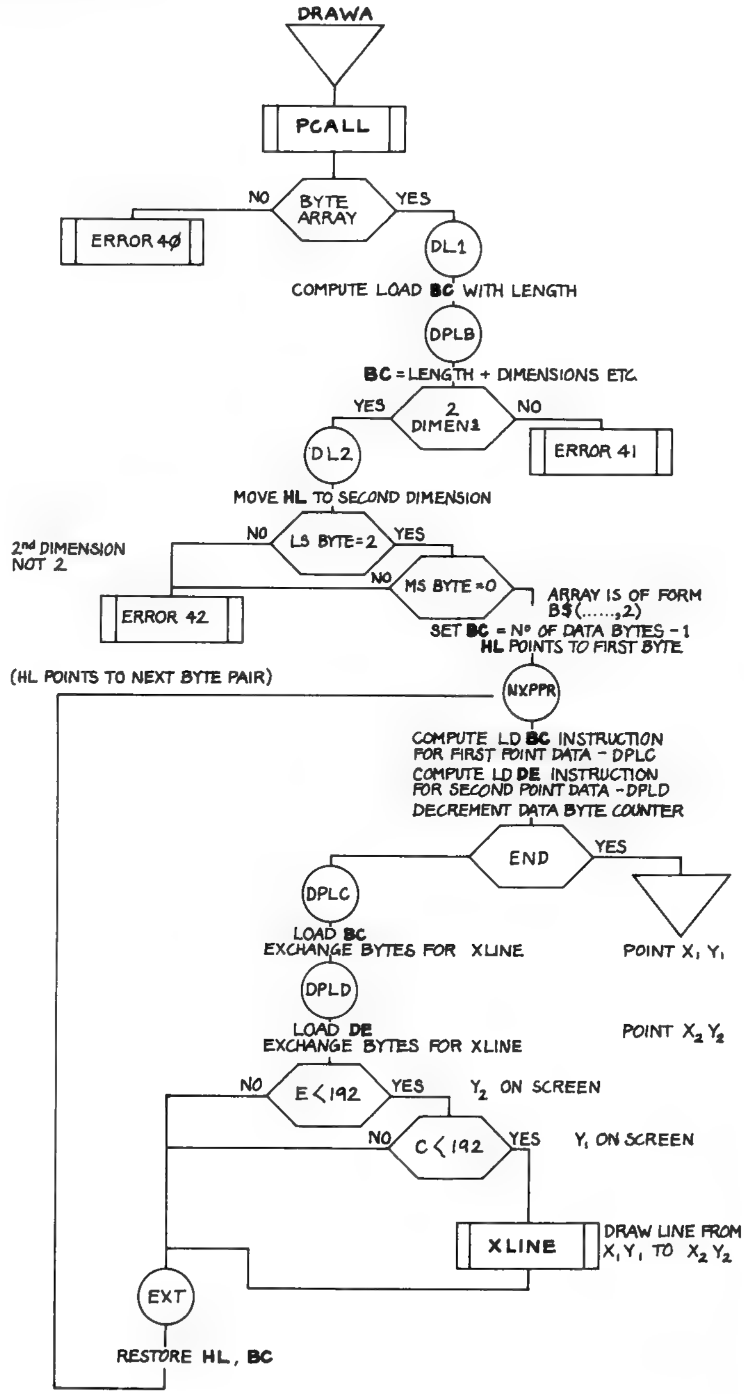
BC and DE, once set up, are checked to ensure that they are both on screen. If either is off the screen the line drawing routine XLINE is omitted and the next point pair is obtained for as long as data remains as tested for by BC greater than zero.

BASIC drawing program

This program using only MOVEC and DRAWA routines enables the drawing of quite complex figures. The keys operate as specified for MOVEC; 'p' causes the cursor position to be transferred into l and hence to the kth slot of k$( ) , a repeated point causes the off screen marker to be inserted and the cursor may then be moved to the head of the next desired line.

I leave you with the problem of how to break out of the drawing routine so that you can save k$( ). Hint: you might reserve the bottom of the screen for a menu of some sort.

Flowchart 14.3



Synopsis

DRAWL allows you to draw a series of connected lines from point 1 to point 2 to point 3... These points are specified as x, y pairs in the REM parameter statement which may be of any length, eg REM : "x1, y1, x2, y2, x3, y3, x4, y4,... xn, yn,".

DRAWA is similar to DRAWL but the data should be supplied in a character array of xy pairs. Points outside the display area are not plotted so lines may be broken by inserting an 'off screen' point in the array.

MOVEC uses the 5, 6, 7, 8 keys to move a cursor around the screen. The 'p' key causes the routine to exit with the current position of the cursor; the x, s and f keys allow single step, slow and fast cursor movement.

A BASIC drawing program is listing in **Listing 14.4**; this is for you to elaborate as you wish.

CHAPTER 15
Miscellaneous

Here are some tit-bits which are nice to know or think about but do not warrant a chapter to themselves.

BCD or Binary Coded Decimal

A form of number representation and arithmetic, believed to be of American origin and dubious parentage. It enabled a salesman to say to a prospective victim: '''but our machine can do decimal arithmetic — you shouldn't bother with one of theirs. Their's can only do (nasty, complicated, difficult) binary'''.

Each decimal digit can be represented by four bits with the values 8, 4, 2, 1 in 8421 BCD. (There is another form 4421 BCD). The Z80 chip will handle BCD arithmetic at two digits per byte if, after each addition or subtraction you insert a DAA operation (Decimal Arithmetic Adjust) and write a special number print routine.

I regard the presence of BCD within a machine as something best overlooked; however, many pieces of electronic equipment do make available BCD coded signals, four wires per decimal read out digit, so that they may be interfaced with computer systems.

Modifications

All I have been able to do, in this book, is point you in the proper direction. No book is ever going to solve all your problems for you, but by way of illustration I have included some code extras which I leave you to understand.

Listing 15.1

```
7610 DEMO1   CALL OPARS
7615         CALL PCALL
7620         CALL FIDL1
7625         JP   SATTR+3
7630 FIDL1   LD   HL,(PARMO)
7635         LD   BC,3
```

```
              ADD   HL,BC
              LD    DE,VPARO+8
              LDI
              RET
     DEMO2    CALL  PCALL
              CALL  FIDL2
              JP    SATTR+3
     FIDL2    LD    HL,(PARMO)
              LD    BC,8
              ADD   HL,BC
              LD    DE,VPARO
              CALL  LDPR
              CALL  LDPR
              CALL  LDPR
              CALL  LDPR
              CALL  LDPR
              RET
     LDPR     LDI
              LD    BC,4
              ADD   HL,BC
              INC   DE
              RET
              END
```

DEMO1

This enters SATTR after the call of OPARS and PCALL. The REM statement it expects is:

REM k: 0, 0, 15, 7,

where k is an (integer) attribute and the constants are a tile region descriptor.

DEMO2

This also enters SATTR but its REM statement is:

REM a( ):

and the first five entries in a( ) are the tile descriptors and the required attribute. These must all be integers.

Both use fiddle subroutines. Note how simple they are, work out how they operate, and have fun doing your own.

Multiple entry

With a large suite of programs a very nasty state of affairs can occur:

Program A outputs to display 1
Program B outputs to display 2

There is a common subroutine C, deep in the depths, doing the actual display output.

A is outputting to the display when the display goes faulty and reports to C, which outputs an error message to the operator and waits for the display fault to be cleared.

Program B now outputs, using common subroutine C, and promptly fouls everything up something rotten unless C is specially written to take care of the problem.

The usual technique is first to estimate the number of multiple calls that can be running at the same time, add 50% (or more) and then set up that number of 'pages', perhaps using the IX register or its equivalent, for all the workspace needed for one entry. Each cell is then allocated a 'page' which is released when that call terminates. If no room is available the calling program must be informed so that it can wait or whatever until the call can be accepted.

Recursion — or flying the Ouzlum bird

Recursion is the calling of a subroutine by itself. This may happen by accident in large programs or be deliberate as a result of a quest for reduced code or otherwise. It almost always demands large amounts of stack space.

Ordinarily, the call of itself will destroy the workspaces and return address, so the subroutine must be deliberately designed to cope with this. In some ways the problem is similar to that of Multiple Entry but here the data is all stored on the stack for entry and a section of the stack is used for workspace as well. The basic technique is illustrated in **Figure 15.1**. You must ensure that the subroutine call on itself must be conditional and that the condition fails so the subroutine can exit and thread its way back to the outside world. If you do not the system, like the Ouzlum bird, will have a nasty accident.

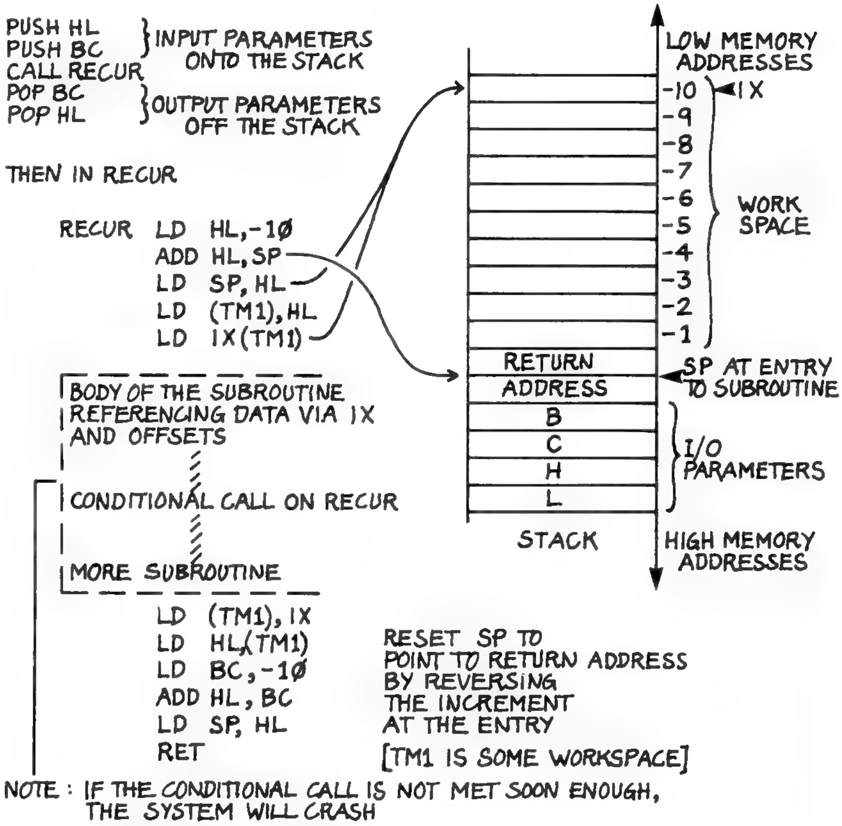
Notes on the machine code and the assembler

All the mnemonics for the operation codes are as standard. The 'hidden' operations, ie those for which the hardware operates but whose existence is not official, are used.

The directive, assembler driving, mnemonics which are used are:

DEFB	defines one byte as a decimal number or ASCII character
DEFS	defines a series of BYTES by using an ASCII string

Flowchart 15.1



DEFW	defines a word of two bytes
END	specifies the end of the machine code
EQU	requires a label, which is assigned the value in its address field. This is usually the address of a Spectrum system variable.
ORG	specifies the head address of the assembled code

A single byte value may be specified by a decimal value (0–255) or an ASCII character enclosed within double quotes. Note that LD A, ““”” loads A with the ASCII code for “”.

Code — do's and don'ts

Assemble the code to run at high memory addresses but leave enough room between the end of your code and the Spectrum UDG pointer location for the stack (see Spectrum manual Chapter 24 page 165) ie at the high address end of WORKSP. In general you will be alright if the end of your code is at about 63500 with a 48K machine.

Never use absolute addresses (numerical values) within your code. Absolute addresses should only be used when addressing Spectrum variables, (as detailed in Chapter 25 pages 173–176 of the manual) or specific parts of the ROM.

Keep notes on all your programming, and your errors!

Make all names as mnemonic as you can.

Write straightforward programs whenever you can. (A program which works is better than none at all, and few drivers ever look under the bonnet.)

Have a very clear idea of what you want to do before you start.

INDEX

The select index is a guide to the main discussions of these subjects. The figures given are for the *first page(s)* on which entries appear.

Printed in England by Commercial Colour Press, London E7.

machine code applications for the ZX spectrum

expert machine code techniques

david laine

Spectrum Machine Code Applications contains advanced machine code routines to deal with problems such as floating point numbers, output to the screen and animated graphics. All the routines are fully explained and annotated.

Through the application of the host of routines presented the author explains how successful machine code routines are written, tested and used in practical applications.

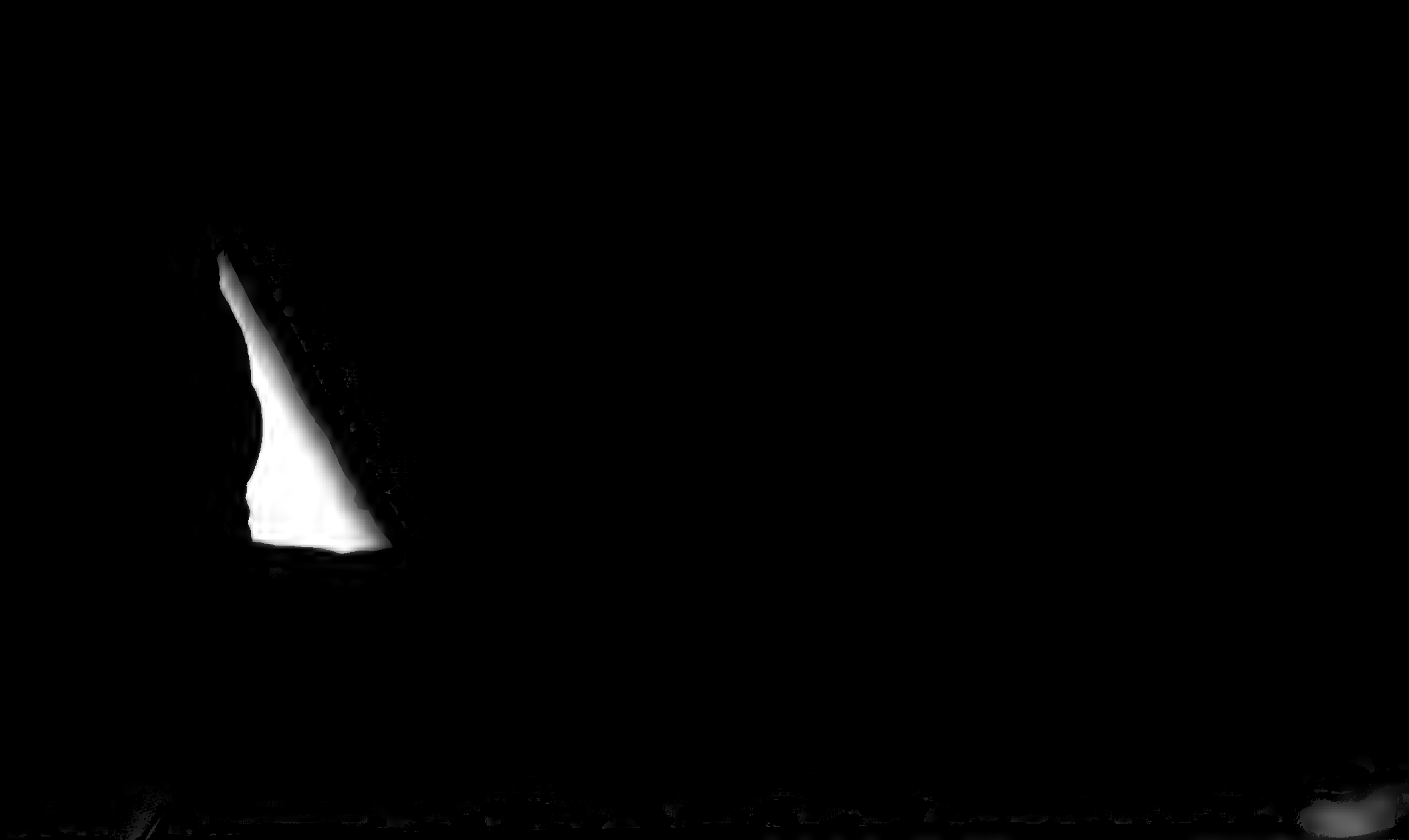
This is not another introductory book on machine code but an insight into the way a professional machine code programmer looks at the Spectrum.

0946408173

SUNSHINE

ISBN 0 946408 17 3

£6.95



DAVID LAINE SPECTRUM MACHINE CODE APPLICATIONS SUNSHINE

machine code applications for the ZX spectrum

expert machine code techniques

david laine

machine code applications for the ZX spectrum

expert machine code techniques

david laine

Spectrum Machine Code Applications contains advanced machine code routines to deal with problems such as floating point numbers, output to the screen and animated graphics. All the routines are fully explained and annotated.

Through the application of the host of routines presented the author explains how successful machine code routines are written, tested and used in practical applications.

This is not another introductory book on machine code but an insight into the way a professional machine code programmer looks at the Spectrum.

0946408173

SUNSHINE

ISBN 0 946408 17 3

£6.95

DAVID LAINE

SPECTRUM MACHINE CODE APPLICATIONS

SUNSHINE

applications for the ZX spectrum

expert machine code techniques

david laine